Broadview®
www.broadview.com.cn

精细化解析京东商城基础架构建设

京东 京东商业技术丛书

京东基础架构建设之路

京东商城基础架构部 著

THE WAY WE WIN

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

京东商城基础架构部

京东商城基础架构部是京东商城的一级部门，专注于核心基础技术的自主研发与工程实施，涉及的技术有数据中心集群管理、数据库系统与分布式存储、电商中间件技术、商城整体架构提升、机器学习与知识工程，为618和双11大促提供强有力的技术支撑，是京东商城的技术基石。

欢迎扫码关注团队公众号
与作者共同探讨交流

京东 京东商业技术丛书

京东
基础架构建设之路

京东商城基础架构部 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

本书内容涵盖容器集群、数据库、分布式存储、服务框架、消息队列、异地多活、机器学习等一系列经典技术话题，深入浅出地向读者展示了京东基础架构的搭建、演进、变革及发展的完整画像，系统地阐述了京东重要阶段的技术进步历程及里程碑级别的技术突破，堪称是一部“从入门到精通”的基础架构经典教材。

作为过去几年里推进京东基础架构变革的技术实践者，我们乐于把自己的经验分享给更多的基础架构从业者或感兴趣的人，技术无止境，愿我们一路相伴共创奇迹。

图书在版编目（CIP）数据

京东基础架构建设之路 / 京东商城基础架构部著. — 北京 : 电子工业出版社，2017.11
（京东商业技术丛书）
ISBN 978-7-121-32865-7

Ⅰ. ①京… Ⅱ. ①京… Ⅲ. ①计算机网络管理-架构-研究 Ⅳ. ①TP393.071

中国版本图书馆CIP数据核字（2017）第244100号

策划编辑：符隆美
责任编辑：张春雨
印　　刷：北京千鹤印刷有限公司
装　　订：北京千鹤印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　　邮编：100036
开　　本：787×980　1/16　　印张：11　　字数：226.5千字
版　　次：2017年11月第1版
印　　次：2017年11月第1次印刷
定　　价：79.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

编委会

序一

这本书在与大家见面的时候，又到了一年一度的双11大促时期，在屏幕前翘首以盼的消费者，摩拳擦掌的商家，蓄势以待的友商，一切都为了那激情燃烧的狂欢。巨大的期待带来的不仅仅是巨大的销量，同样也会给电商系统带来巨大的压力。辉煌的背后，无不是坚守在系统一线技术人员的殚思极虑。伴随着市场的繁荣及消费者的巨大差异化，我们持续通过技术创新来提升用户体验，努力让购物从简单的交易变成贴心的帮助与关怀。

基础架构是京东业务的技术基石。为了不断提升系统的稳定性，我们自主研发了容器计算平台JDOS、分布式存储JFS、弹性数据库JED，以及JIMDB、JMQ、JSF等中间件服务；在今年，我们还顺利实施了异地多活，即商城广域分布架构，初步建立了机器学习与知识图谱等数据算法类服务。在系统压测方面，京东以往大促前主要通过订单后军演等方式进行多次压测，但毕竟无法准确模拟真实场景；从今年的618开始，我们大规模应用“军演机器人”系统，真切地模拟大促开始时海量订单涌入的情景，让每个系统都得到充分压测，有效发现性能瓶颈。

本书的作者，作为过去几年里推进京东基础架构变革的技术实践者，一直坚定执行京东集团“下一个十二年，只有技术”的发展路线，并在这条路上不断迎接新的挑战，为提升京东商城的用户体验不断努力。

我也借此书，代表所有奋战在一线的技术研发团队，将京东在基础架构技术领域这几年中的发展和创新分享给关注我们的朋友。感谢所有互联网技术从业者对我们的关注、帮助与指正。我们会持续推进技术架构演进，为京东电商业务继续高速发展保驾护航。

马　松

京东集团高级副总裁、商城研发体系负责人

序二

从人类历史的发展来看，最终影响人们生活、推动社会进步的是科技的发展。每个历史时代都会涌现新的科技，使人类社会不断向前进步。那么什么是科技？其实我们可以将其定义为需求驱动的以基础科学研究与突破为依托的解决产业问题的科技体系。就本书而言，京东的基础架构就是本着以解决问题为导向，同时通过技术的积累与创新，采用技术驱动业务的途径逐步实现的。

本书的主要作者刘海锋是我十多年前的学生，从他的书中能读出他在这些年的技术积累与沉淀，同时也读出了我们中国科学技术大学校训中“理实交融”的思想——理论研究需要与实际应用联系在一起；也再一次印证了中国科学技术大学注重基础研究并以解决实际问题为导向的学风、多元交叉的院系设置和学科设置对解决产业问题的极大帮助。

《人类简史》让我们看到人类的发展是跟文明的进步和科技的变革息息相关的，而《未来简史》将这几年的热点词——“人工智能”“大数据”——纳入了人类历史中，给出了一些历史命运层面的抽象，有追捧也有争议。那么，在这个大家热议的人工智能时代，也使我们有了更多的反思，其实在这些热点议题的背后，一些基础架构与底层系统技术的发展与实现或许更加务实和接地气一些，同时产业界也需要有更坚实的基础架构与底层系统技术来支撑日益增长的庞大的业务量。而《京东基础架构建设之路》这本书，从底层的容器管理集群技术，到服务框架、分布式内存数据库和分布式文件存储系统，再到机器学习在京东的多场景应用和商品数据知识图谱的构建，都做了详细的介绍，向大家展现了整个系统搭建的发展历程。同时，书中也解密了京东技术研发在每年618和双11超大流量和高并发时刻的应对策略，相信会对互联网和电商行业的从业者有着不错的借鉴作用。

陈恩红

中国科学技术大学计算机学院教授、博士生导师、副院长

前言

很幸运地在2013年5月加入京东，更加幸运地遇到了一批优秀的同事，在接下来的4年半的时间里，兄弟们并肩作战，自主研发并持续建设了一系列核心基础架构系统。我们将这几年的工作成果及经验整理总结，写在书中分享出来，希望能给大家带来收获！

第1章“容器集群技术”，主要介绍数据中心操作系统JDOS，即容器集群管理平台。经历过物理机管理分配的各种痛苦后，我们在2014年8月启动了JDOS项目，做了一个简单、勇敢、并不艰难的决定：跨过VM时代，直接基于Docker做容器！而当时大部分公司是通过虚拟化技术提供私有云服务的。不走寻常路，使得我们团队在容器集群技术领域走在前面。从最初很小的规模，到管理几乎所有服务器；从仅调度应用容器，到统一集成中间件、存储和数据库服务；从很像VM的“胖容器”，到应用集群编排；从非常像IaaS，到更接近PaaS……总之，这4年的容器化集群管理技术实践，让我们亲眼见证、亲手推动了一系列变革。JDOS团队这4年只做这一件事情，一件事持续做，就做得很不平凡。JDOS团队当前着力推进“阿基米德项目”，即融合计算资源管理、统一在线与离线混部，预期在2018年全面上线并创造巨大的收益。

第2章“数据库技术”，主要介绍了弹性数据库系统的演进。历经SQL Server、Oracle、物理机部署MySQL，从2015年起我们逐步以“MySQL in Docker”作为数据库服务的主流方式。京东很可能是第一家普遍采用容器化MySQL交付的国内互联网公司。2016年秋，我们在此基础上立项实施弹性数据库项目（简称JED），目标是实现关系型数据库的弹性扩展能力。JED可以称为一套fully cloud-native NewSQL：运行于JDOS 2.0之上，完全容器化部署；具备scale up/down （通过容器技术本身）+ scale out/in（通过过滤复制协议）两种弹性伸缩能力，并且支持服务器与数据中心两个级别的扩展性；基于MySQL作为其存储与复制引擎，且兼容MySQL协议便于应用接入。JED站在JDOS肩膀上，吸收了京东文件系统JFS、缓存平台JIMDB等分布式存储系统的设计经验。从今年（2017年）618之后，JED就成为京东所有应用默认的数据库服务。

第3章“分布式存储技术”，主要介绍京东文件系统JFS及商品图片系统。JFS为解决海量小文件的分布式存储问题而生，亦为之量身定制。2013年我初到京东，当时订单文本、物

流报文等BLOB主要存储在数据库中，导致扩展性受限只能定期删除；商品图片服务由一套开源系统搭建，稳定性与性能都不甚理想。于是披星戴月地开发了第一个系统——JFS，服务端用Go语言，客户端用Java语言，并在当年10月上线，陆续承载了OFC订单履约、WMS库房报文等业务的海量数据。2014年年初，基于JFS的新图片系统成功上线——按照历史图片迁移、数据校验、流量切换三步走，历时一个半月，顺利完成切换。我至今记得切换完成的那天晚上，跟几位兄弟击掌相庆的情景，胜利的喜悦总是让人记忆犹新。之后，JFS增加了大文件存储、元数据管理及S3兼容等功能；图片系统则在质量优化、透明压缩等方面不断提高。四年间，商品图片数目从数亿一路增长至数百亿，而JFS与图片系统一直稳定如初。

第4章“中间件技术”，主要介绍以服务框架JSF、消息队列JMQ、缓存平台JIMDB为代表的中间件技术体系。对国内任何一家电商公司来说，中间件系统都是一块核心的技术基石。质量则是中间件技术的生命线。我们在2014年年初自主研发服务框架JSF、消息队列JMQ、缓存平台JIMDB。每个系统差不多都是开发测试一年，推广完善一年。自2016年开始，整个中间件体系步入稳定期，使得业务开发团队能够完全专注于产品功能实现。中间件系统的自主可控能力，加上全托管式服务，大幅提升了产品研发效率。JSF几乎部署在京东的每个IP上，被每个Java应用程序所import；JMQ是订单、支付、履约、物流的数据管道，也是每次大促前军演性能比赛的第一名；JIMDB则从无到有，逐步发展到今天多个IDC数千台大内存服务器的部署规模，并存储京东商城几乎所有的动态内容——JIMDB是世界级的memory-as-the-new-disk NoSQL服务。

第5章“整体架构升级”主要介绍了“ForceBot：全链路军演机器人”和“异地多活”两部分内容。

◎ForceBot：全链路军演机器人。每年618、双11两次大促的技术备战，是京东研发团队的重要工作，而其中一个关键环节就是做充分的、全面的压力测试来有效地发现性能瓶颈并指导资源规划。各个服务的独立压测既不能模拟真实线上的流量状况，也不能全链条覆盖各个环节；往年常用的订单后军演仅覆盖到订单后系统，而大促流量压力主要在订单前。基于数次大促备战经验，我们在2016年着手建设ForceBot，即全链路在线军演压测系统，并推动各个应用系统与基础系统改造来支持“forcebot=1”在线压测流量标识。ForceBot通过部署在各地CDN节点的机器人程序来模拟海量用户的实际行为，包括登录、浏览、搜索、点击、加购物车、下单提交、支付等购物全流程，向京东公网入口发起对应千万用户同时在线的巨大访问流量。2016年双11，ForceBot小试牛刀，初见效果；2017年618即全面实施，成为大促前备战工作的重要组成部分，也是资

源规划与性能优化的主要手段。ForceBot使得如今的大促备战智能化、常态化。

◎广域分布架构，俗称“异地多活”。不同公司、不同技术团队，对所谓“异地多活”可能有不同理解。“异地多活”这个形象的词汇，描述的是工程结果的表象，并非动机或者目标。从我们的思考角度，做异地多活，主要出发点是什么？公司持续增长的各项业务使得我们需要不断去寻找更大的机房，但这在同一地区并不现实。核心目标是什么？实现数据中心粒度的容量扩展，即通过增加IDC提供资源弹性。相比之前的增加服务器来扩展某个系统，这是分布式架构的更高级形式：多地域多IDC分布式架构；从数据中心角度，实际上是多个不同地域、不同规格的数据中心，形成一个逻辑上更大的数据中心。至于更强的容灾能力和用户就近接入，则是异地多活的附加收益。在切身经历同城单机房、双机房、数次扩容机房之后，我们在2017年年初开始做项目规划，然后进行系统设计、技术改造，并在2017年10月完成第一期实施。异地多活架构升级，我们实施得并不算早，但是采用了与同行不同的技术方案：完全在中间件层面做改造，对应用系统完全透明。当然，弹性数据库JED也是实现商城广域分布架构的核心组件。

第6章“机器学习技术”主要介绍京东商城基础架构中的机器学习与知识工程。我个人非常相信：AI = Machine Learning + Knowledge Engineering。机器学习与知识工程会成为新的基础设施、所有应用共享的技术服务。自2016年8月，我和团队一起，积极开展针对商品图片的视觉计算相关研究、开发与应用；并着力建设针对商品数据的京东知识图谱，向上提供Knowledge as a Service优化原有业务并赋能创新产品。JDOS的大规模计算资源调度能力，特别是CPU/GPU统一服务，为机器学习、深度学习、数据挖掘和多模态数据清洗提供了强有力的底层支撑。此外，我们正在研发新一代图数据管理与计算系统——“波特”，旨在将知识图谱塑造为强大的知识计算引擎。很多激动人心的项目，都在积极进行之中。

以上所述这些工作，紧密联系、互相支撑，共同组成了京东商城的技术基础架构，大规模部署于京东多个数据中心的数万台物理服务器上，运行着无数的在线业务，并产生、存储、处理着海量的电商数据。在过去四五年的时间里，在不断迭代的研究、设计、开发、测试、维护、优化、升级的过程中，无论是我个人还是团队里的其他兄弟，都深刻感受到技术本身强大的力量。我们对互联网技术的热爱与憧憬，永远如初。

感谢中国互联网，特别是电商业务的高速发展，感谢京东集团过去几年里梦幻般的发展速度，给了我们这些技术从业者最好的舞台。感谢我们的家人、公司领导与同事的大力支

持。最想感谢的，是奋斗在第一线的所有京东研发人员！

本书撰写在日常工作之余，时间仓促，不准确或遗漏之处在所难免，欢迎各位读者的批评与反馈。里面所介绍的各个项目或者系统，仍然在持续建设之中，后续的新成果，我们会通过各种方式与本书的读者分享和交流。

刘海锋

京东商城总架构师、基础架构部负责人

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误**：您对书中内容的修改意见可在 *提交勘误* 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 *读者评论* 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：*http://www.broadview.com.cn/32865*

二维码：

目录

01 第1章 容器集群技术

1.1 概述 3
1.2 JDOS 1.0：“胖容器”时代 3
1.3 JDOS 2.0：新一代应用容器引擎 9
1.4 JDOS 3.0：服务融合平台 15
1.5 JDOS 4.0：弹性数据计算 18
1.6 总结 22

02 第2章 数据库技术

2.1 发展历程 25
2.2 BinLake日志订阅服务 27
2.3 弹性数据库 33

03 第3章 分布式存储技术

3.1 JFS：京东文件系统 47
3.2 JIMDB：内存是新的磁盘 52
3.3 FBase：大表存储 60
3.4 Container File System 66

第 4 章
中间件技术

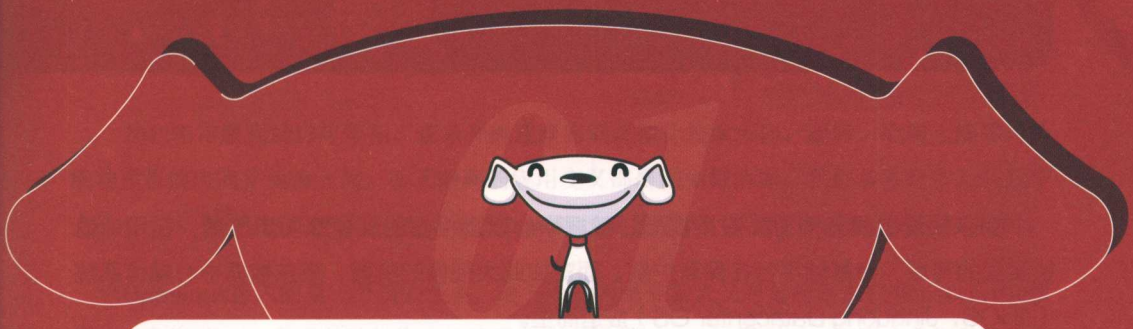
4.1 服务框架 75
4.2 消息队列 88
4.3 JMQ复制技术解析 101
4.4 CallGraph：分布式服务跟踪系统 112

第 5 章
整体架构升级

5.1 ForceBot：全链路军演机器人 125
5.2 异地多活 133

第 6 章
机器学习技术

6.1 基于机器学习的商品数据治理 145
6.2 智能分单 156
6.3 列表页排序 157
6.4 语音识别与客服导航 159
6.5 商品上新助手 162



第 1 章
容器集群技术

1.1 概述

1.2 JDOS 1.0：“胖容器”时代

1.3 JDOS 2.0：新一代应用容器引擎

1.4 JDOS 3.0：服务融合平台

1.5 JDOS 4.0：弹性数据计算

1.6 总结

京东快速发展的同时，应用规模、数据中心以及机器的规模都同步倍增，在用户愉快买卖的背后，有基础平台在保驾护航。面对如此大规模的机器，京东数据中心操作系统（JDOS，Jingdong Datacenter OS）应运而生。

JDOS统一管理调度京东海量计算资源，服务应用快速交付，从容支撑大促高峰流量。

历经4年时间的技术沉淀与发展，JDOS不仅仅作为京东数据中心操作管理资源，更作为京东统一的PaaS平台，致力于支撑业务系统快速交付、稳定运行，基础中间件托管提升基础平台敏捷交付，以及正在演进中的融合计算项目。

本章会与大家分享JDOS团队的发展历程、业务服务现状和技术发展阶段。在内容呈现上，本章讲述了从JDOS 1.0容器化“胖容器”开创京东容器化技术之路，到完成京东全部业务容器化运行后，随即推动JDOS 2.0，完全基于镜像发布方式，建设完整的容器技术生态的内容。自研分布式智能DNS、高性能负载均衡，以及为容器量身打造的分布式共享存储ContainerFS，这些都是为了更好地建设JDOS与数据中心协同发展的生态。在完成JDOS实现数据中心操作系统赋能后，进而推动JDOS 3.0以构建京东的PaaS平台，从源码、编译、构建镜像、容器集群编排、副本控制、日志、监控、中间件能力等完整PaaS生态。随着JDOS规模持续增大，精细化运营促使启动了正在演进中的JDOS 4.0阿基米德项目融合计算，实现在线业务与离线大数据计算混部，智能调度，并在满足业务计算需求的同时节约采购成本。在容器技术实践方面，京东是发展较早也比较坚定的，在实践过程中有很多理解和技术感悟。借这本书的机会，很高兴能够跟更多的业界同仁分享JDOS的经验，希望对业内有所启迪。

1.1 概述

2012年年底到2013年年初，京东开始逐渐布局虚拟化技术方向。那时，应用上线等待分配物理机的时间平均为一周，而无隔离的应用交叉部署则如履薄冰。我们从零起步，从几个人的小团队开始起航，逐渐搭建起专业的OpenStack研发团队，掌握了OpenStack的核心代码。

OpenStack接入的第一个核心业务，是一个并发量非常大又对延迟要求在40ms以内的0级系统。我们对KVM做了各种优化，依然无法达到预期。2013年夏到2014年夏，我们只能退而求其次，在KVM环境中支撑了几百个非核心系统运行。这一年，有郁闷，有压力，但也在积攒经验；这一年，团队对公司业务有了深刻的理解，对OpenStack做到了熟练的定制开发。

2014年秋，公司布局私有云技术方向，团队重新出发。经过仔细调研与思考，我们做了一个大胆的决定——跳过虚拟化，直接容器化！那时候的Docker非常单薄，单薄到只有镜像和对cgroup简单的操作等功能是生产可用的。另外，那时并没有成熟稳定的容器管理系统。考虑到团队积累的OpenStack经验，我们选择了OpenStack + Docker的架构，即用OpenStack来管理调度Docker容器——如虚拟机般的容器，戏称为“胖容器”。

我们将平台命名为JDOS，即Jingdong Datacenter OS的缩写。也就是说，从项目启动之初，JDOS的愿景就不仅仅是要做容器管理平台，而且要做数据中心的操作系统！

1.2 JDOS 1.0：“胖容器”时代

1.2.1 集群管理

集群架构

京东第一代容器引擎（JDOS）1.0版本从2015年开始部署，并在当年10月陆续将部分业务迁移上来。第一批业务包括一系列核心系统，如单品页、图片处理、订单等。

JDOS 1.0基于OpenStack Icehouse + Docker 1.3 + OVS 2.1.3，架构简单可靠。如图1-1所示，为单集群部署架构。为了支持万台服务器的规模，我们对OpenStack代码，特别是对其中

的消息队列与DB依赖部分进行了架构优化。

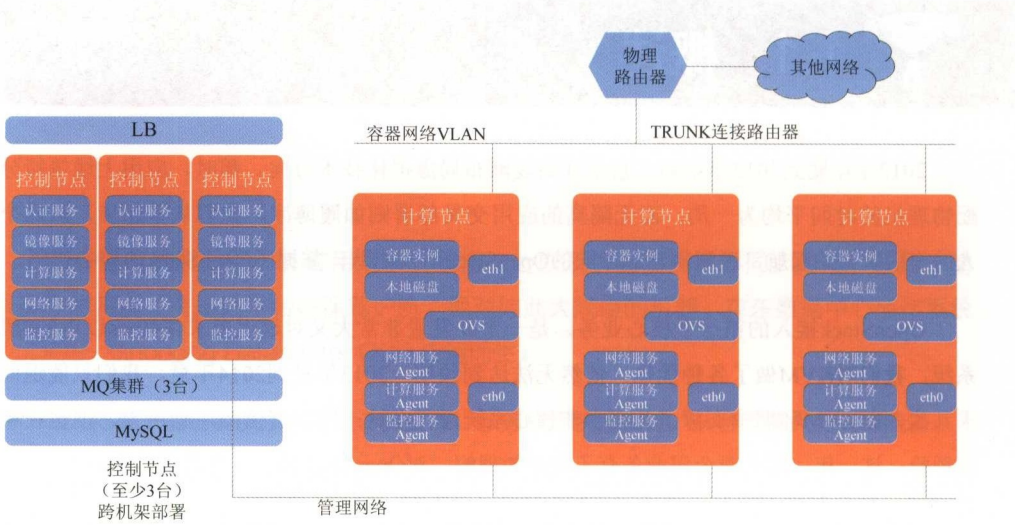


图1-1　单集群部署架构

我们采用多IDC部署方式，使用统一的全局API开放对接到上线系统。满足应用跨机架、跨物理机器、跨网络Pod分配容器资源，更好地支持高可用应用设计。

由于业务需求多样，集群功能迭代会产生多个版本，并且会在线上共存。我们需要对每个服务进行精确的管理。通过给每个服务增加一个版本号的标记能够有效解决这些问题。

根据业务的特性，我们制作了一系列的基础镜像。在每个数据中心部署镜像仓库，在客户端配置多个数据中心仓库地址（优先选择本地数据中心）。常用基础镜像会预分发到各个计算节点。

容器集群的运维

我们维护的单个OpenStack集群有1万台物理计算节点，最小的有4千台计算节点。京东容器化战略实施的这几年，运维是其中很大的挑战。在JDOS研发之初，即把可运维性定为核心目标，因此JDOS运维工程师从未超过2人！日常运维工作系统归类如下。

◎平台容量扩展。一套基于Chef的自动部署，在大促前集中上线扩容的时候核算过，从机器上架加电完成后开始计算，到新的节点加入集群资源池为止，可用的效率是 4千台物理节点/天/人。

◎物理机硬件故障。京东统一监控平台（MDC）也是基础架构部设计研发推出的，它有

着全新的设计、跨IDC、基于容器部署、监控效率高效且故障信息可以自动收敛等优势，特别是对硬件故障（网卡CRC错误、内核信息、ILO）的感知及时而准确。我们还特别与机器学习团队合作，对硬件故障进行智能预测，特别是对磁盘故障的预测收获极大。这些信息都会自动通知机房现场的IDC同事进行处理，并自动通知受影响业务方，同时给出预测恢复时间。

◎每日巡检。从物理机、OS、OpenStack、依赖的组件、内核日志到进程，会进行全面巡检，如图1-2所示。

XX机房2区	
巡检项目	状态
Rabbitmq集群状态	Running Nodes: rabbit@MQNODE1 rabbit@MQNODE2 rabbit@MQNODE3 Rabbitmq consumers are ok!
系统日志级别检查	All controll and compute nodes system log level are info!
系统日志错误节点检查	Host: node1 NIC Link is Down Host: node2 NIC Link is Down
所有节点ntq时间误差检查	Host: node3 TimeError: 39s Host: node4 TimeError: 48s
数据口网卡异常节点	All compute nodes network are OK!
数据口网卡crc错误检查结果	Host: node5 Crc Error: 8
LVM Config检查结果	All compute nodes' lvm config are OK!
nf_conntrack检查结果	Host: node6 nf is open
TC检查结果	All compute nodes don't have tc module!
arp检查结果	All compute nodes arp are in normal state!
容器swap检查	Host: node7 has container with wrong swap value
网卡中断均衡检查结果	All compute nodes smp_affinity are ok!
文件句柄数检查	All compute nodes' file descriptor are greater than 20480000!
JDOS团队	

图1-2　巡检邮件

1.2.2 资源监控

监控模块是集群中不可或缺的重要模块之一，承担着监控指标、数据展示、告警订阅、通知等重要功能，为日常维护、定位问题、压测等场景提供数据支撑。从监控目标来看，容器集群中的资源监控可分为容器监控和集群内的物理机监控。我们通过自研的MDC系统来承担集群的资源监控任务。

MDC（Monitor Data Center），是京东自研的企业级监控系统，承担着服务器及容器的监控任务。系统提供海量历史数据查询，方便问题回溯，并有实时监控、统计报表等功能，方便跟踪定位疑难问题、压测观察、统计分析等方方面面的研发场景，支持用户订阅通知服务，可通过多种渠道给予用户及时、准确的通知服务，为应对紧急情况赢得时间。

规模方面，MDC监控数据包含系统指标、网络指标、磁盘指标、硬件指标、容器指标五大类指标，共31个小类常用的指标监控服务。每天生产监控数据数百亿条。正如图1-3所示，系统的总体架构功能划分清晰，将采集与任务管理解耦。任务可以无状态地在Agent间迁移，方便横向扩展。

整体结构自上而下可分为如下四大部分。

◎Console，负责数据配置、展示等环节，为用户使用平台提供界面和入口。

◎Controller，对内负责采集资源的管理和采集任务的调度，对外提供丰富的数据获取，报表生成接口。

◎Agent，监控Agent，包含采集、告警、数据处理等多个模块。

◎存储（如图1-3所示的Cache和HBase），负责监控数据的存储。

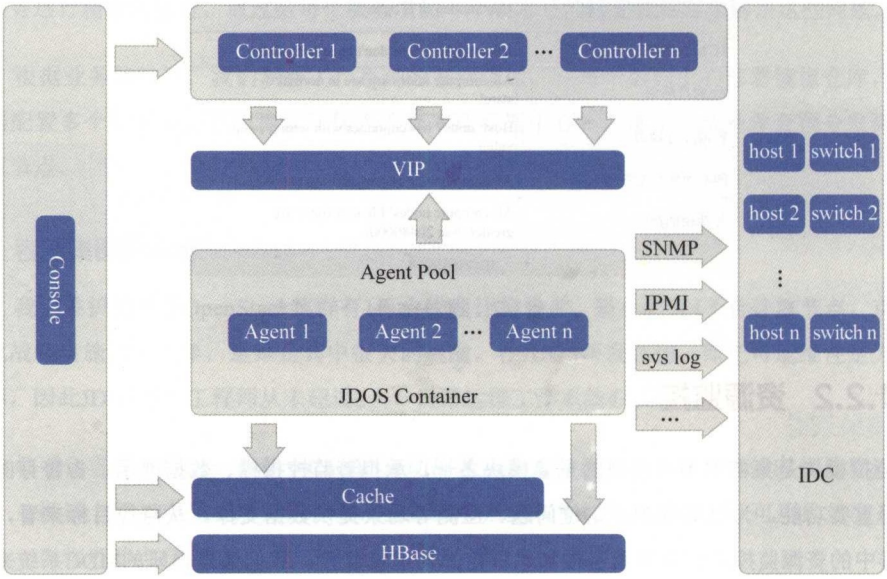


图1-3　MDC架构

1.2.3 容器定制

容器与虚拟机

当向别人讲述什么是容器时，常常用虚拟机做类比。容器可以理解为一种轻量级的虚拟机。但是用户也需要明白容器与虚拟机这两者是有根本区别的。

虚拟机本质上是模拟。通过模拟物理机上的硬件，向用户提供诸如CPU、内存等资源。因此虚拟机上可以且必须安装独立的操作系统，系统内核与物理机的系统内核无关。因此，一台物理机上有多个虚拟机时，一个虚拟机操作系统的崩溃不会影响到其他虚拟机。而容器的本质是经过隔离与限制的Linux进程。容器实际使用的还是物理机的资源，容器之间共享了物理机的Linux内核。这也就意味着当一个容器引发了内核crash之后，会殃及物理机和物理机上的其他容器。从这个角度来说，容器的权限和安全级别没有虚拟机高。但是反过来说，因为能够直接使用CPU等资源，容器的性能会优于虚拟机。

容器之间的隔离性依赖于Linux提供的namespace。namespace虽然已经提供了较多的功能，但是，系统的隔离不可能如虚拟机那么完善。一个最简单的例子，就是一台物理机上的不同虚拟机可以设置不同的系统时间，而同一台物理机上的容器只能共享系统时间，仅仅可以设置不同的时区。另外，对于容器资源的限制是通过Linux提供的cgroup来实现的。在容器中，应用是可以感知到底层的基础设施的。而且，由于无法充分隔离，从某种程度上说，容器可以看到宿主机上的所有资源，但实际上，容器只能使用宿主机上的部分资源。

graph driver

在容器化落地的实践过程中，难免会遇到很多坑。其中一个坑就是graph driver的选择问题。当时的Device Mapper（DM）驱动，在频繁读/写的情况下，会遇到内核crash的问题。这个问题在当时比较棘手，我们一开始没能解决，于是自己写了一版graph driver，命名为vdisk。vdisk主要是通过稀疏文件来模拟union filesystem的效果。这在实际使用中会减慢创建速度，但是益处是带来了极高的稳定性，而且不再有DM的data file预设磁盘容量的限制了。因为容器创建后，还需要启动公司的工具链、运维确认、切流量等才能完成上线，所以在实际中，该方式有着极好的效果。

不过我们没有放弃DM，之后通过配置nodiscard参数解决了DM带来的内核crash问题。配置nodiscard参数虽然解决了内核crash的问题，但是在实际的实践中，又引入了一个新的问题，这就是容器磁盘超配的问题。比如，DM的data是一个稀疏问题，容量为10GB，现在有5个容器，每个容器磁盘预设空间是2GB，但是容器文件实际占用只有1GB。这时，理论上来

说，创建第6个2GB的容器是没有问题的，但是虽然这5个容器的实际文件可能只占用1GB，可如果容器中对于某个大文件进行反复的创建、删除（如Redis的aof），则在DM的data中，会实际占用2GB的空间，这时第6个容器就会因为DM的data空间不足而无法创建。这个问题从DM的角度来说不好直接解决，在实际操作中，可以通过与业务的沟通，将这种频繁读/写的文件放置到外挂的volume中，从而解决这个问题。

我们给用户建议，镜像和根目录尽量只保留只读和少量读/写的文件，对于频繁读/写或大量写的文件，尽量使用外挂的volume。当然，对外挂的volume也做了容量和写速度的限制，以免业务之间互相影响，规范业务对容器的使用行为。

Docker版本

在开始研究Docker时，Docker的稳定版本还是1.2和1.3。但是随着Docker的火热发展，版本迅速迭代，诚然，新的版本增加了很多新功能，但是也可能引入了一些新bug。而且，当一台宿主机上的Docker需要升级时，容器也需要迁移，那么可能涉及多个业务方的沟通协调。我们逐渐定制了自己的Docker版本，并稳定使用。Docker提供的功能足够，尽量不再升级，新增功能可以通过其他的方式进行实现。

1.2.4 标准化镜像

在没有容器化之前，公司内部已经形成了一套工具链体系，诸如编译打包、自动部署、统一监控等，并为线上应用提供了稳定的服务。因此，在JDOS落地实践中，我们兼容公司已有工具链。在制作镜像时，将原有的工具链也都打入了镜像中，真正实现了业务的平滑迁移。

当然，打包了诸多的工具，随之而来的就是镜像的臃肿，镜像的体积也不可遏制地从几百MB增长到了GB级别。借助于工具链的标准化，镜像的种类就被缩减为了几种。考虑到创建容器的速度，我们采用镜像预分发的方式，将最新版本的镜像及时推送到计算节点上，虽然多占用了一些磁盘空间，但是有效防止了容器集中创建时，镜像中心的网络、磁盘读/写成为瓶颈的问题。

1.3 JDOS 2.0：新一代应用容器引擎

1.3.1 1.0的痛点

JDOS 1.0解决了应用容器化的问题，但是依然存在很多不足。首先，编译打包、自动部署等工具脱胎于物理机时代，与容器的开箱即用理念格格不入，容器启动之后仍然需要配套工具系统为其分发配置、部署应用等，应用启动的速度受到了制约。其次，线上线下环境仍然存在不一致的情况，应用运行的操作环境、依赖的软件栈在线下自测时仍然需要进行单独搭建，线上线下环境不一致也造成了线上问题难于在线下复现，更无法达到镜像的“一次构建，随处运行”的理想状态。再次，容器的体量太重，应用需要依赖工具系统进行部署，导致业务的迁移仍然需要工具系统人工运维去实现，难以在通用平台层来实现灵活的扩容、缩容与高可用。最后，容器的调度方式单一，只能简单地根据物理机剩余资源来进行筛选调度，在提升应用的性能和平台的使用率方面存在天花板，无法做更进一步的提升。

1.3.2 平台架构

鉴于以上不足，在JDOS 1.0容器数目逐渐增长到10万左右规模时，即2016年年初，我们启动了新一代容器平台——JDOS 2.0——的研发。JDOS 2.0的目标不仅仅是一个基础设施的管理平台，更是一个直面应用的容器引擎。JDOS 2.0在原1.0的基础上，围绕Kubernetes，整合了JDOS 1.0的存储、网络，打通了从源码到镜像，再到上线部署的CI/CD全流程，提供从日志、监控、排障、终端、编排等一站式的解决方案。如表1-1所示，为各功能的技术选型列表。

表1-1　各功能的技术选型

功　　能	选　　型
容器工具	Docker
容器网络	Cane
容器引擎	Kubernetes
镜像中心	Harbor
持续集成工具	Jenkins

续表

功　　能	选　　型
日志管理	Logstash + Elasticsearch
监控管理	Prometheus

在JDOS 2.0中，我们定义了系统与应用两个级别：一个系统包含若干个应用，一个应用包含若干个提供相同服务的容器实例。一般来说，一个大的部门可以申请一个或者多个系统。系统级别直接对应于Kubernetes中的namespace。同一个系统下的所有容器实例会在同一个Kubernetes的namespace中。

应用不仅仅提供了容器实例数量的管理，还包括集群版本管理、域名解析、负载均衡、配置文件、日志服务、操作历史等服务。不仅仅是公司各个业务的应用，大部分的JDOS 2.0组件也实现了容器化，在JDOS 2.0平台上进行部署。如图1-4所示，为JDOS应用管理的界面。

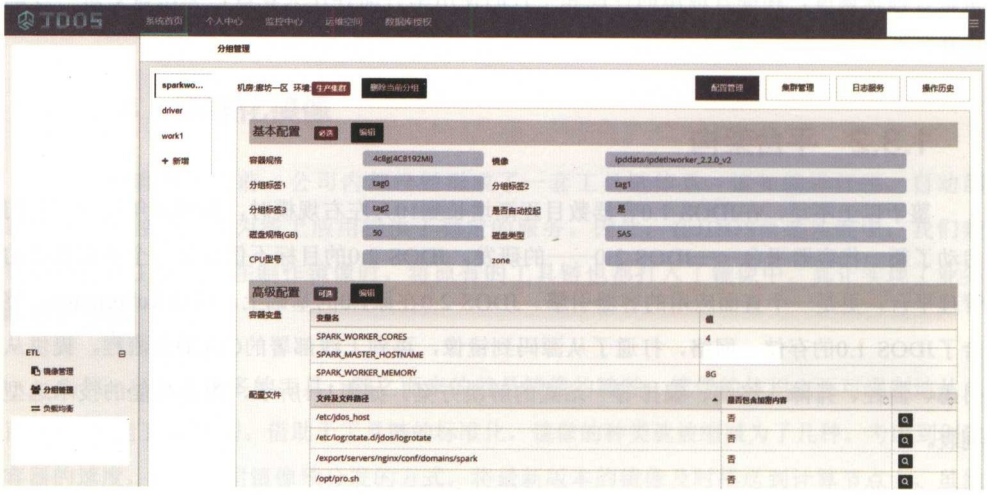


图1-4　JDOS应用管理的界面

1.3.3　开发者一站式解决方案

JDOS 2.0实现了以镜像为核心的持续集成和持续部署，如图1-5所示。

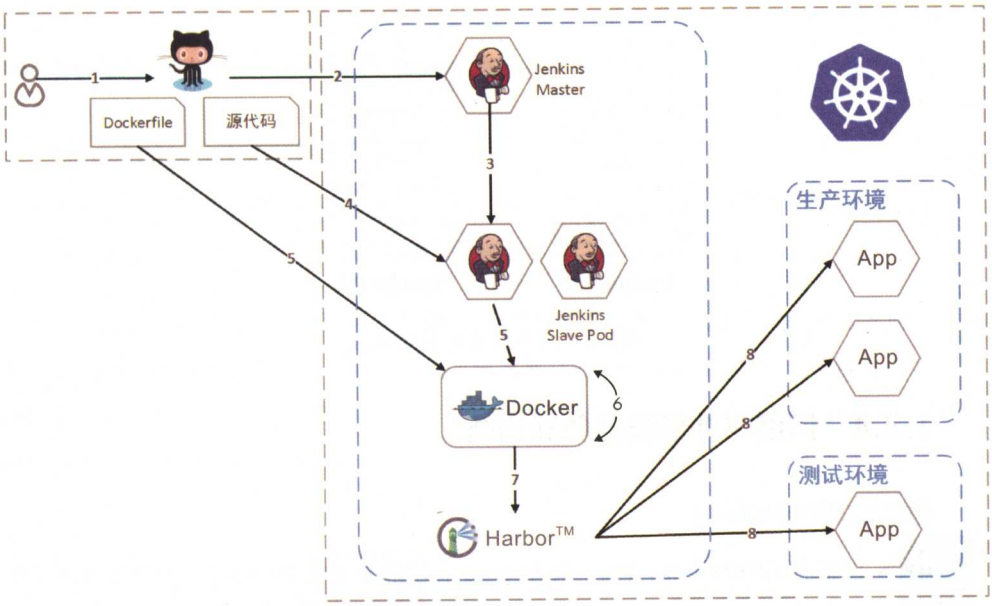


图1-5 持续集成和持续部署

持续集成和持续部署的主要步骤如下。

1. 开发者提交代码到源码管理库。

2. 触发Jenkins Master生成构建任务。

3. Jenkins Master使用Kubernetes生成Jenkins Slave Pod。

4. Jenkins Slave拉取源码进行编译打包。

5. 将打包好的文件和Dockerfile发送到构建节点。

6. 在构建节点中构建生成镜像。

7. 将镜像推送到镜像中心Harbor。

8. 根据需要在不同环境生产、更新应用容器。

在JDOS 1.0中，容器的镜像主要包含了操作系统和应用的运行时软件栈。App的部署仍然依赖于以往运维的自动部署等工具。在JDOS 2.0中，我们在镜像的构建过程中完成应用的部署，镜像包含了App在内的完整软件栈，真正实现了开箱即用，一次构建到处运行的愿景。如图1-6所示，为两个版本的镜像比较。

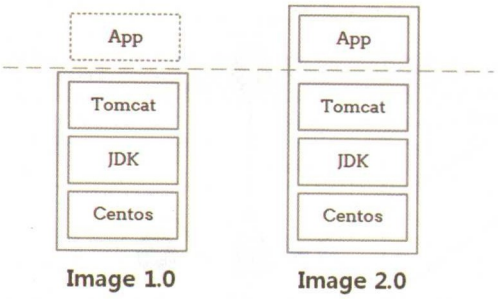


图1-6　两个版本的镜像比较

1.3.4　网络与外部服务负载均衡

稳定高效的容器网络

JDOS 2.0继承了JDOS 1.0的方案，采用OpenStack-Neutron的VLAN模式。该方案实现了容器之间的高效通信，非常适合公司内部的集群环境。每个pod占用Neutron中的一个port，拥有独立的IP。基于CNI标准，我们开发了新的项目Cane，用于将kubelet和Neutron集成起来，如图1-7所示。

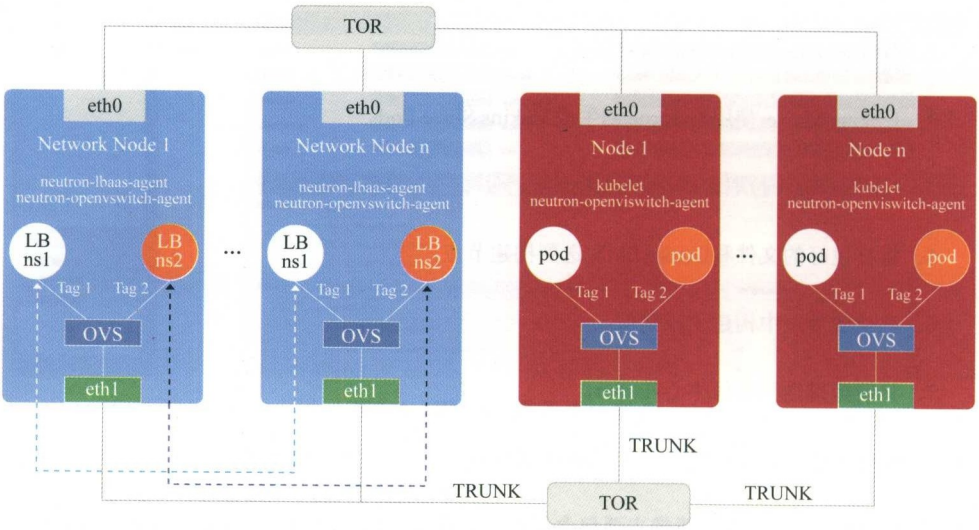


图1-7　网络与负载均衡

同时，Cane负责Service中LoadBalancer（LB）的创建。当有LoadBalancer类型的Service创建、删除、修改时，Cane将对应地调用LBaaS服务中创建、删除、修改LB的服务接口，从

而实现外部服务负载均衡的管理。另外，Cane项目中的“SkyDNS”组件为容器提供了内部的DNS解析服务，“SkyLB”组件提供高性能负载均衡服务。

SkyDNS：分布式智能DNS服务

作为数据中心最基础的核心服务之一，SkyDNS具有以下特点：支持自动发现服务域名、支持智能后端探活、横向扩展、容器部署、智能解析。

架构方面，解决传统DNS配置文件reload生效机制采用etcd watch模式，自动实时发现域名变更，并将DNS服务组件完全解耦，使之成为独立的API、DNS解析、后端服务探活、智能IP库自动更新等进程。SkyDNS与JDOS 2.0平台密切配合，完成Kubernetes的Service发布。SkyDNS架构图如图1-8所示。

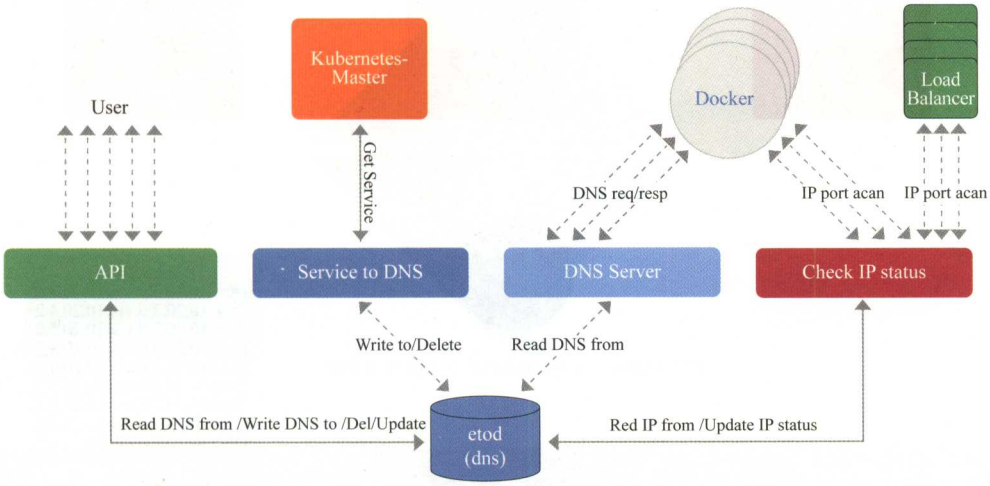


图1-8　SkyDNS架构图

SkyLB：高性能负载均衡服务

用户可以在JDOS平台创建负载均衡来获得SkyLB提供的服务。SkyLB是基于DPDK实现的高性能负载均衡服务。目前，支持的负载均衡调度算法有一致性Hash、Round Robin和最小连接数算法。SkyLB核心架构是通过Session五元组来实现会话的管理功能的。SkyLB Session如图1-9所示。

SkyLB使用BGP模式组成集群，通过BGP协议发布VIP，交换机将数据包散列到集群中各个节点上，保证单台SkyLB故障或者恢复后能动态地将机器添加或删除。其冗余实现设计如

图1-10所示。

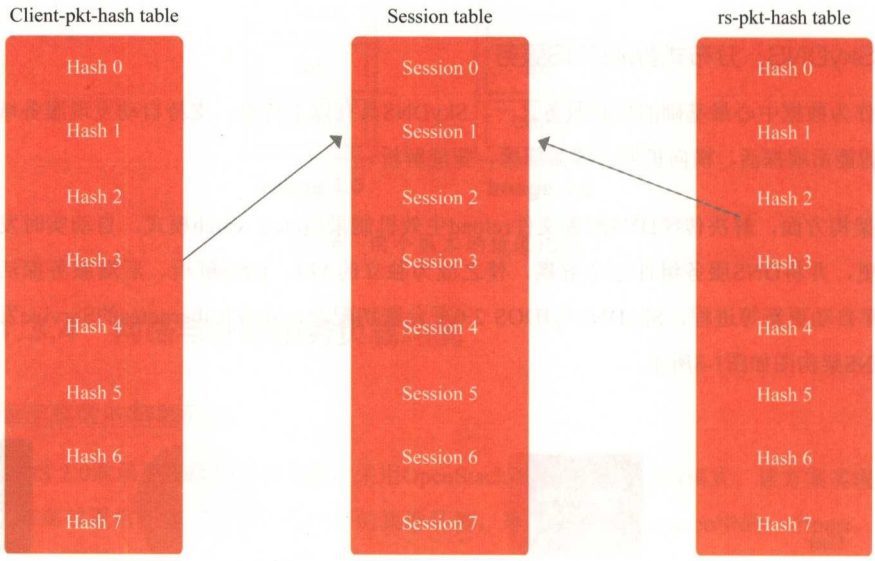


图1-9 SkyLB Session

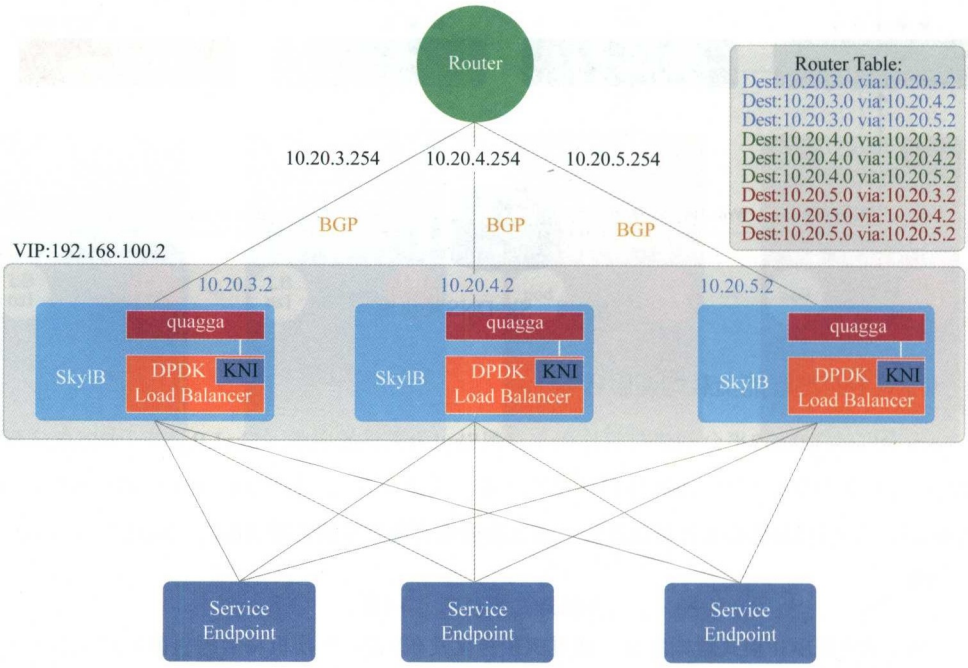


图1-10 SkyLB冗余实现设计

强悍的转发性能，借助DPDK的优势，如便利的多核编程框架、巨页内存管理、无锁队列、无中断poll-mode 网卡驱动、CPU亲和性等来实现快速的网卡收发及处理报文。在普通x86服务器上基准测试10Gbps NIC http 120W QPS，结果贴近网卡的硬件性能极限。

1.3.5　灵活调度

JDOS 2.0接入了包括大数据、Web应用、深度学习等多种类型的应用，并根据类型为每种应用采用了不同的资源限制方式，且打上了Kubernetes的不同标签。基于多样的标签，我们实现了更为多样灵活的调度方式（如图1-11所示），并在部分IDC上实验性地混合部署了在线任务和离线任务。相较于1.0，2.0的整体资源利用率有显著提升。

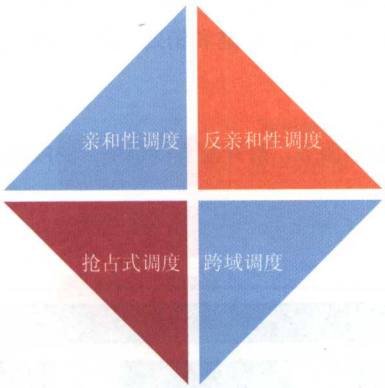


图1-11　调度方式

1.4　JDOS 3.0：服务融合平台

1.4.1　平台架构

JDOS 2.0很好地解决了用户需求，以面向应用为目标，打通了从源码到镜像再到上线部署的CI/CD全流程，提供从日志、监控、排障、终端、编排等一站式的功能。如果应用比较独立，依赖的服务又很少，那么用JDOS 2.0就足够了。如果应用依赖很多外部服务，而这些外部服务又不是运行在JDOS 2.0上面的，应用就需要去各个依赖的服务申请入口进行服务使用申请，再回到JDOS 2.0依次添加各个服务的配置。

考虑一下这样的场景：一个应用A，需要依赖很多京东内部的后端服务。那么，A应用的发布部署需要经过以下步骤。

1．应用在JDOS 2.0上面很顺畅地完成了代码编译、构建镜像等操作。

2．开始应用配置这个环节，由于要依赖很多后端服务，这些后端服务资源需要去各个后端服务页面申请。这里开始阻塞了，需要经历一个较长时间的等待过程，直到所有的后端服务资源申请成功，才能开始把应用配置完成。

3．创建应用集群，应用发布成功。

可以看到，该应用的发布痛点体现在上面的第二步，阻塞时间会比较久。虽然基础架构部提供JIMDB、JMQ、JFS、MySQL、JED等一系列后端服务，但是如果需要使用这些后端服务，往往需要走单独的流程进行申请、创建和销毁，需要消耗额外的人工，而且无法实现自动化的生命周期管理。

JDOS 3.0旨在融合各种后端服务，给开发者提供统一的Platform as a Service体验。其架构图如图1-12所示。

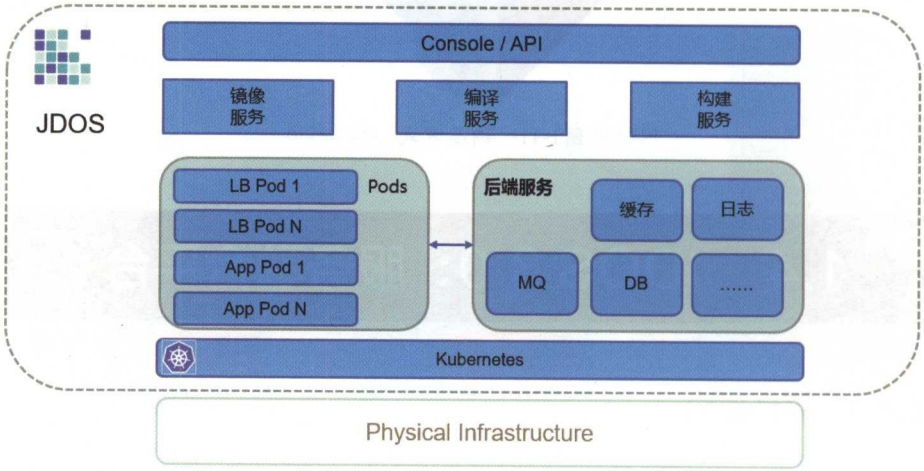


图1-12　JDOS 3.0架构图

从JDOS 3.0的视角看，一切皆资源。根据12-Factor（参考https://12factor.net/zh_cn/）的定义，JDOS 3.0把后端服务（Backing Service）当作附加资源（Attached Resource），与常见的Service、Pod等资源地位等同。JDOS 3.0认为后端服务是可以按需加载、卸载、替换的资源。例如，如果应用的数据库服务由于硬件问题出现异常，那么管理员可以从最近的备

份中恢复一个数据库，卸载当前的数据库，然后加载新的数据库，整个过程都不需要修改代码。

1.4.2　后端服务目录

JDOS 3.0通过标准的Open Service Broker API来集成京东各式各样的后端服务。我们开发了后端服务目录（Service-catalog）项目，通过该项目，用户可以轻松地将其应用程序配置为使用这些后端服务，而无须详细了解这些服务后端真实的创建和管理逻辑。一个新的后端服务只要注册到JDOS 3.0的后端服务目录中，就可以被用户使用了。

在用户上线发布过程中，后端服务目录会列举出所有已经支持的后端服务（例如JMQ、JIMDB等），用户单击某个后端服务，就可以开始按需创建后端服务实例。如图1-13所示，列举了一些后端服务。

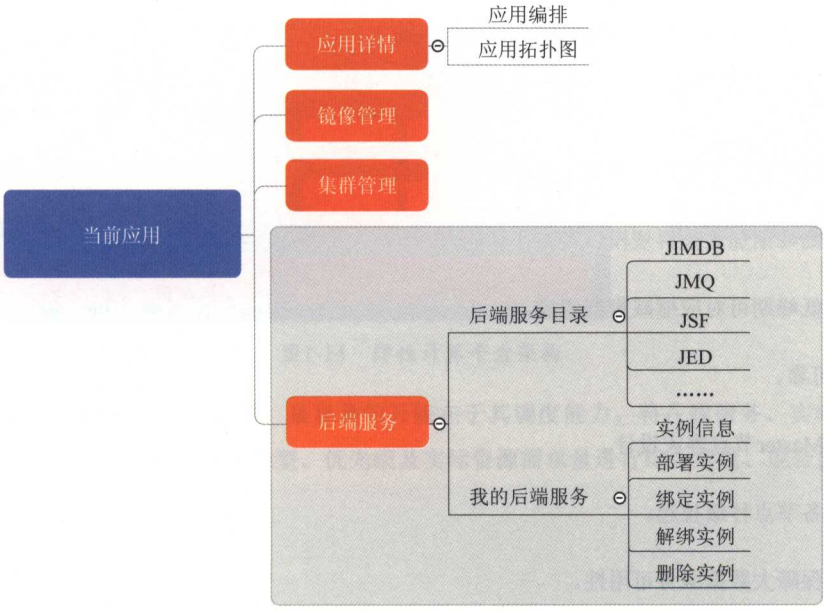


图1-13　后端服务举例

1.5 JDOS 4.0：弹性数据计算

完成JDOS 3.0的建设后，为进一步提升数据中心资源的使用效率，以及助力京东AI场景应用落地和发展，我们设计和研发了弹性数据计算平台，即JDOS 4.0。弹性数据计算平台是构建在JDOS 2.0和3.0的基础上的，利用Hadoop、Spark、Storm、Flink等开源大数据生态系统，为用户提供安全、低成本、高可靠、可弹性伸缩的全托管计算集群，提供集群、作业、数据等管理的一站式大数据处理分析服务。弹性数据计算平台相较于原来的数据平台，有以下优势。

◎便捷。

○一分钟内即可获得一个安全可靠的集群。

○无须为节点分配、部署程序、优化投入时间。

◎弹性。

○创建任意大小的集群并动态调整集群规模。

○高峰期加大集群规模以提高计算能力。

○低峰期可对应缩减集群规模。

◎可靠。

○Master节点容灾设计。

○备节点秒级拉起。

○保障大数据服务可用性。

○完善的监控体系建设。

◎开放。

○完全兼容开源Hadoop、Spark等。

○零成本业务迁移。

1.5.1 平台架构

弹性计算平台为用户提供了相互隔离的计算资源集群，并可以根据实际的计算需求和集群状况进行统一的调度。用户可以根据实际的计算需要，选择使用Storm、Flink、Spark、Hadoop等计算框架进行计算。弹性计算平台支持从HBase、MySQL等公司的公共数据源平台获取数据，同时也支持用户将数据传入弹性计算平台的存储中。弹性计算平台的存储会挂载到对应集群的容器内，使得计算资源集群能像使用本地数据文件一样使用数据。弹性计算平台架构如图1-14所示。

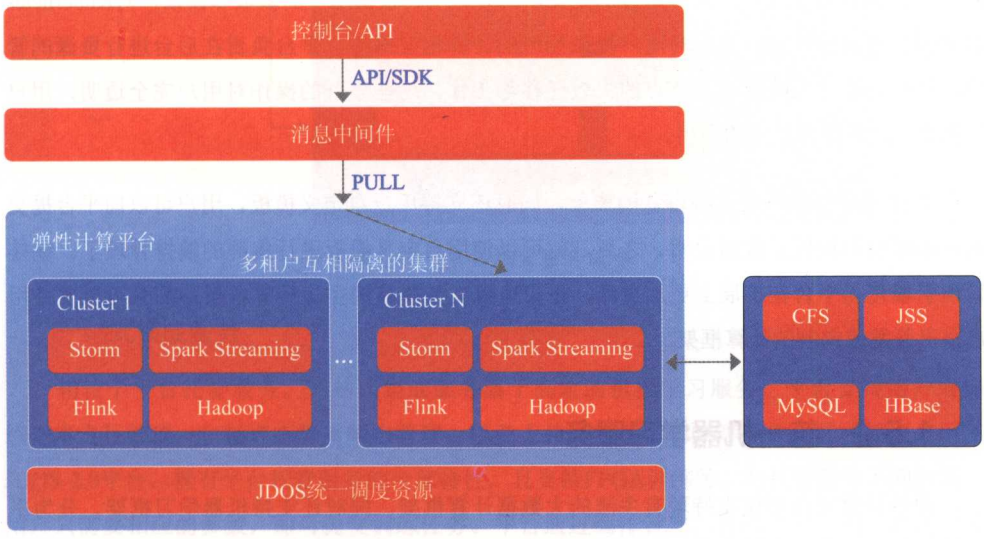


图1-14 弹性计算平台架构

JDOS 4.0相比之前的版本，最显著的升级在于其调度能力。将在线服务、实时计算任务、离线计算任务，根据任务类型、优先级及实际资源需求量进行综合考量、混合部署并弹性调度。

GPU支持

弹性计算平台将更多种类的计算资源纳入平台进行统一管理。平台不仅支持传统的CPU、内存等，还支持GPU的统一管理。根据用户的申请，弹性计算平台将GPU分配给用户，并挂载到容器中。

CPU的管理模式是可以复用的，即多个用户的容器可以使用同一个CPU。而GPU的管理模式与CPU的管理模式不同，因为GPU不仅包含计算资源，还包含了显存。由于目前Linux还

没有类似于cgroup的工具对GPU资源进行管理限制，且两个容器同时对GPU和显存进行资源竞争时，会导致应用的崩溃等问题，因此我们对GPU的管理进行了简化，将GPU资源进行独享模式分配，即同一个GPU只能分配给一个容器，一个容器可以分配多个GPU。

编排管理引擎

弹性计算平台支持多个计算框架的核心是编排管理引擎。JDOS 4.0内置Hadoop、Spark、Flink、Storm等多个框架的模板。模板定义了构建一个计算框架集群所需要的Kubernetes的资源情况，诸如使用多少个Pod，Pod的镜像、环境变量等参数的定义，以及使用多少个Service，Service的端口转发等参数的定义，等等。平台根据用户的参数和模板，对Kubernetes的资源进行编排管理，从而向用户提供相应的计算框架服务。平台负责在后台进行资源的管理，包括容器的失败拉起、节点的失效迁移等工作。这些集群的操作对用户完全透明，用户只需要专注于计算框架的使用。

编排管理引擎不仅支持内置的模板，同时还支持用户自定义模板。用户可以向平台提交对应的模板和模板元数据说明。之后，就可以使用自定义模板进行集群的编排管理了。编排管理引擎使得平台在扩展上更加灵活，不仅可以支持现有的主流计算框架，而且可以支持用户自己开发或定制的计算框架。

1.5.2 统一机器学习服务

弹性计算平台不仅支持现有主流的大数据计算框架，同时还支持机器学习框架。分为平台托管和一站式集成两种服务方式来供用户进行选择使用，并提供模型验证、模式发布等一站式解决方案。

平台托管服务

弹性计算平台提供了包括TensorFlow、Caffe、Spark MLlib在内的多个机器学习框架镜像。用户通过平台根据需要选择镜像创建机器学习的容器；容器可以根据需要选择配置GPU资源；平台根据用户选择创建对应的Pod、Service等资源；用户在容器中进行独立管控机器学习训练的启动、停止等；平台提供日志查看、监控数据查看功能。如图1-15所示，为平台托管服务流程。

平台托管服务适合对机器学习框架进行深入的代码改造，或对机器学习的分布式训练进行特殊定制的各AI团队。平台托管服务属于低阶的机器学习服务模式，需要用户对所使用的框架有较深入的理解。

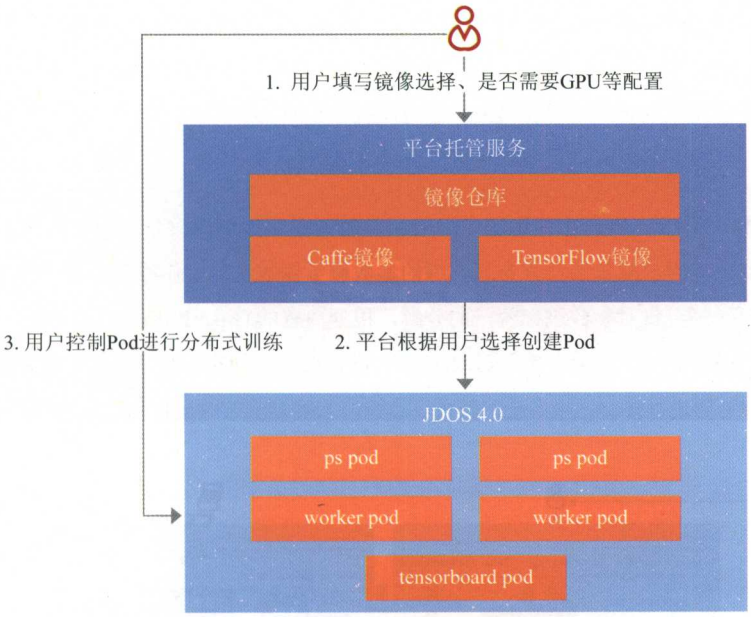


图1-15 平台托管服务流程

一站式集成服务

相较于平台托管服务，一站式集成服务属于高阶的机器学习服务。平台支持用户对训练脚本进行管理，包括训练脚本版本管理，以及上传、修改、删除等操作。其在底层复用了JDOS 2.0平台，拥有平台托管服务的全部特性，且支持FPGA训练等。与托管服务不同的是，用户只需要相应的参数，即可提交训练任务。平台通过编排管理，自动将训练的容器进行分配部署，无须用户管理，即可运行分布式机器学习训练。同时，平台提供了实时的训练进度查看功能。如图1-16所示，为一站式集成服务架构。

图1-16 一站式集成服务架构

一站式集成服务提供了包括数据处理、机器学习、深度学习、模型验证、模型发布等服务。平台内置了多种数据处理和机器学习算法。用户可以将数据处理和机器学习的算法步骤通过页面拖动形成有向无环图（DAG），提交给统一机器学习平台。平台将首先对DAG进行分解，并生成相应的代码，最终提交到Spark集群进行计算。

如图1-17所示，用户在使用深度学习服务时，需要提供深度学习训练的脚本，并上传到平台的存储中。用户配置参与分布式训练的参数服务器数量、工作服务器数量等参数，将参数提交给平台。平台将对参数进行统一的分解，根据内置的TensorFlow、Caffe模板生成对应的容器、负载均衡等资源，并最终提交到JDOS平台进行相应的创建。用户自己无须分别对参数服务器、工作服务器进行控制，平台会对其统一管控，当容器失效时会自动进行迁移重建。

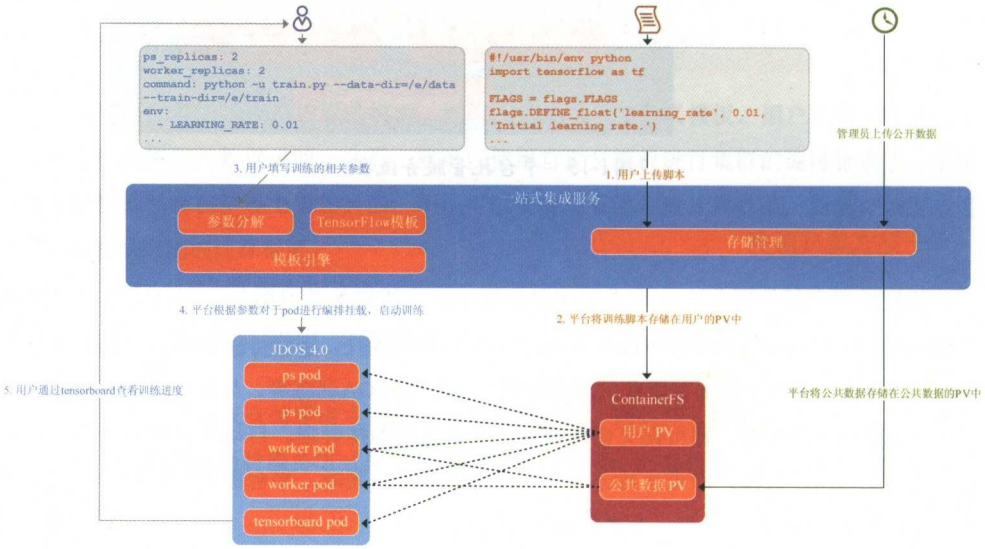


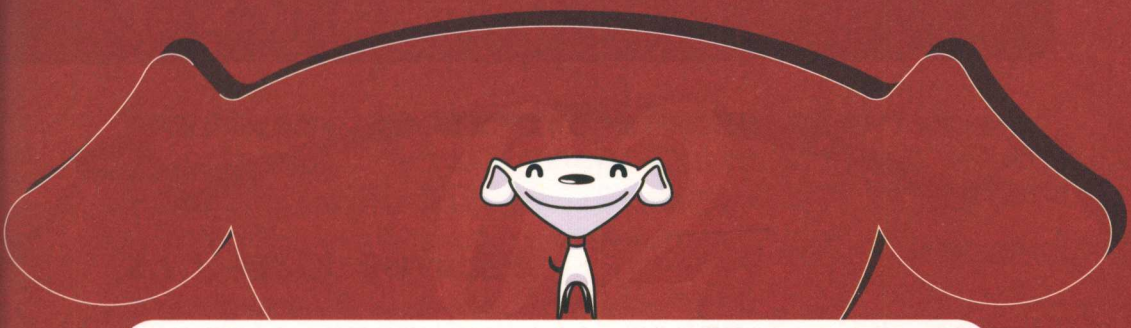
图1-17　一站式集成服务流程

一站式集成服务对于用户的使用门槛较低。用户无须关心实际训练的部署步骤和情况，而只需要将精力集中在训练脚本的开发中。

1.6　总结

从JDOS项目启动至今，一做就是三年。三年做一件事，自然有了些许收获。

目前我们正在进行JDOS的进一步设计开发——内部代号为“阿基米德项目”。敬请期待！



第 2 章
数据库技术

2.1 发展历程

2.2 BinLake 日志订阅服务

2.3 弹性数据库

什么才是互联网时代的数据库架构模式？DBA在未来技术发展中的角色和定位如何？在过去的十三年里，为了满足京东业务的迅速发展，数据库架构也在不断地改进和优化。数据库类型也经历了从商业数据库到开源数据库（例如MySQL、MongoDB等），再到自主研发的JED弹性数据库的转变。一路走来，数据库规模不断扩大，高并发、高可用架构也由单机房部署到多机房部署并实现异地多活，数据库架构由传统主从发展为可动态扩展的弹性数据库，数据库运维工作由手动运维发展为自动化、智能化运维。

本章详细记录了京东在这方面的尝试和思考，其核心观点简单明了，京东未来的技术架构是构建在弹性数据库基础之上的，可以快速响应业务的需求，有效利用资源、降低成本、提升数据库性能。这意味着DBA的角色和知识体系结构必须发生变化，DBA除了具备基础的数据库运维技能，还需要对数据领域各个产品和技术有所涉猎，包括数据库核心技术、数据库自动化运维技术、故障自愈技术、数据安全与审计、数据同步与传输和数据处理等，从而促进京东数据库架构向更全面、更合理的可持续性方向发展。

2.1 发展历程

伴随着京东业务的高速增长，数据库技术选型也一直在进化，前后经历了SQL Server、Oracle、MySQL三个时代。

2.1.1 与SQL Server结缘

在京东发展之初，代码开发以.NET方向为主，因此采用了Windows + IIS + ASP.NET + SQL Server 的组合方式。当时几乎所有的数据存储均由SQL Server承载，京东的第一笔订单数据就写入了SQL Server的数据库中。SQL Server复制与SQL Server集群技术先后在生产环境中应用。

2.1.2 Oracle篇

随着公司业务的爆炸式发展，系统越来越多，架构越来越复杂，从系统性能、稳定、容量、运维和生态圈等多个维度考虑，.Net + SQLServer + Windows的架构已经不能完全满足公司当时的开发和运维需求。Java逐渐在研发系统中占据了主流，与Java开发相配合，Oracle作为最成功的商业数据库，被引入京东并陆续应用在一些核心系统上，比如订单中间件、OFC（订单履约）、青龙配送、POP第三方平台等。Oracle部署在IBM小型机上，主库采用RAC以保证高可用，备库采用DataGuard进行跨机房灾备部署。

系统在上线后的头两年，确实满足了京东业务快速发展的需要，618及双11大促都平稳度过。但是随着数据量和业务量的持续快速增长，集中式Oracle数据库架构的缺点逐渐凸显——应用相互影响、容量难以扩展，高成本、高维护费和高依赖也进一步制约了其持续发展。

随着普通PC服务器稳定性提高，I/O卡和SSD等硬件设备的技术也进一步成熟。在此背景下，我们着手逐渐推进去中心化与去IOE的项目。

2.1.3 MySQL篇

随着MySQL及NoSQL等开源软件的兴起，基于技术可控和授权成本方面的考虑，京东开始逐渐将业务数据从SQL Server和Oracle向MySQL迁移，仅保留少量业务和第三方应用程序继

续使用SQL Server和Oracle数据库。在整个京东去除商业数据库的过程中，我们也体会到：没有最优的数据库架构，只有更满足业务需求的数据库架构。而SQL Server和Oracle DBA也在这个过程中，逐渐转变为既掌握商业数据库运维，又掌握MySQL运维的复合型DBA，有力地支撑了京东业务系统的发展。

2012年，是京东数据库发展的一个分水岭，MySQL在京东迅速崛起，且爆炸式地增长。到2013年，占京东一半的交易系统使用了MySQL。发展到2014年、2015年，MySQL已逐渐成为京东的主流数据库。随着去IOE项目的推进，SQL Server、Oracle数据库几乎全部迁移到了MySQL数据库。

京东MySQL的5个大事记如下。

1. 2012年的春节，MySQL服务器的数量突破了一百台。
2. 2012年6月，订单中心开始使用MySQL。
3. 2013年，一半以上的手机客户端价格、移动、促销、库存等核心数据服务器使用MySQL。
4. 2014年，特别是在2014年双11大促时，订单系统已经正式使用了MySQL，由此宣布了MySQL在京东的主流地位。
5. 2015年以后，京东90%以上的应用都使用了MySQL。

MySQL数据库在公司的大规模应用，使运维自动化平台建设成为必然。京东从如下4个维度推进了MySQL运维自动化。

◎自动备份。通过灵活的调度策略，在业务低谷时间段将MySQL数据文件与Binlog文件备份至云存储（JFS）上，并支持按时间点的精准恢复。

◎自动历史数据结转。将不再变更的历史数据，定期结转至大容量分布式存储系统（如开源HBase或自研FBase）中，有效控制MySQL数据库的数量。

◎自动故障检测与切换。我们采用Orchestrator作为MySQL HA与复制拓扑的管理工具。

◎全面容器化。随着容器平台JDOS的稳定成熟，2015年我们开始推进MySQL容器化项目，利用容器的镜像管理、快速创建等特性，大幅提升了MySQL实例的交付效率。目前，京东商城几乎所有的MySQL进程都运行在Docker容器中。

2.2 BinLake日志订阅服务

2.2.1 背景

BinLake之前，各个业务部门若想要对数据库的Binlog日志进行采集和订阅，并根据订阅到的Binlog日志进行分析并实现特定的业务目标需要经过以下6个步骤。

1. 日志采集产品调研（例如canal）。

2. 资源申请。

3. Binary Log采集服务部署。

4. 发布/订阅系统调研（例如Kafka）。

5. 日志采集与消息发布/订阅系统集成开发与测试。

6. 业务应用开发、测试、上线。

这种方式既费时又费力，每个部门都需要配备负责日志采集和消息处理的专业工程师，并且都需要从头到尾将整个技术堆栈重新梳理和调研一遍，而且上线后还需要持续地投入人员进行持续的运维和管理。

为了解决上述问题，BinLake应运而生。BinLake是一个平台化的数据库Binary Log管理、采集和分发系统，并且透明集成JMQ和Kafka等消息分发和订阅系统。通过使用BinLake，可以便捷地实现数据库Binary Log的自动化采集、分发和订阅。

2.2.2 架构设计

BinLake的整体架构设计如图2-1所示。

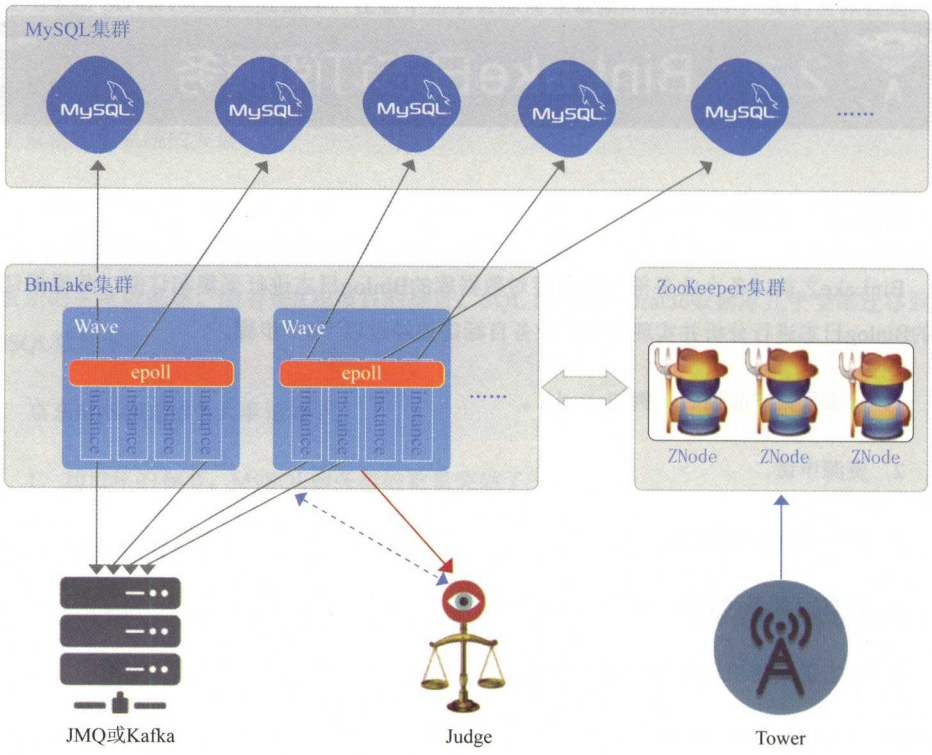


图2-1　BinLake架构图

BinLake总共包括如下三大服务组件。

◎Wave服务。Wave服务完成实际的数据库Binary Log的持续采集、管理和分发，并写入下游的消息发布/订阅系统中。在BinLake集群中会存在*N*个Wave服务，这些Wave服务共同组成一个无状态集群。

◎Tower服务。Tower服务是整个BinLake的管理中心，提供BinLake接入服务的申请、完成Wave服务、数据源、接入应用的管理。当用户申请接入到BinLake中时，会登录到Tower服务提供的申请界面，填写申请接入到的BinLake的应用信息、数据源信息和Topic信息，Tower服务会根据用户提供的信息，并按照图2-2中描述的逻辑完成用户的接入申请。

如果不同申请者申请相同数据源的数据采集，则由Tower管理端依据其申请的采集规则（如指定表、指定库）进行处理，如果规则相同，则默认复用相同规则的Topic，也可强制生成新的Topic进行订阅。

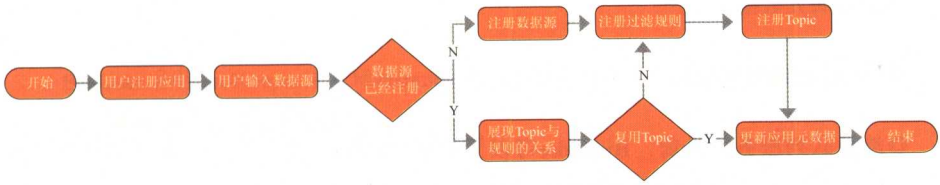


图2-2 BinLake Topic注册逻辑流程

◎Judge服务。Judge服务主要完成两个功能：Wave节点监控信息采集和LoadBalance决策。

○Wave节点监控信息采集：通过在各个Wave服务节点部署Agent，采集各个Wave服务节点上的监控信息，包括服务器的内存使用、系统负载、CPU负载、网络负载、JVM的堆内存使用、GC信息及每个Wave服务中的instance个数等，采集到的所有这些信息都会在后续的LoadBalance中作为基础metics，参与到最终的LoadBalance决策中。

○LoadBalance决策：新应用接入到BinLake时，若需要采集的数据源在BinLake现有的数据源池中不存在，则需要对新的数据源在相应的Wave服务上创建对应的instance（数据源与instance是1对1的关系）。这时，就会请求Judge服务提供的LoadBalance决策接口，若Judge服务中没有配置LoadBalance plugin，则会返回一个随机的Wave服务节点的IP，在该随机的Wave服务上创建instance；若配置了LoadBalance的plugin，则从Judge服务提供的LoadBalance决策接口获得建议的Wave服务节点，并从该节点创建新的instance。

BinLake依赖于如下两大外部服务。

◎ZooKeeper（ZK）。BinLake使用ZooKeeper服务进行Wave无状态集群的管理、状态同步和消息通知等，包括如下7项。

○instance的自动化创建与初始化。

○instance的HA。

○数据源offset实时追踪。

○Binlog分发失败重试。

○数据源切换自适配。

○Tower元数据管理。

○instance消息通知。

◎消息队列服务。目前，BinLake可以无缝集成JMQ（京东自主研发的消息系统）和Kafka，从而进行消息的发布和订阅管理。instance采集到的Binlog Event会发布到JMQ或者Kafka的Topic中，实际的业务应用只需要订阅和消费对应的Topic，实时获得Binlog Event，并在后续的业务逻辑中对获得的Binlog Event进行处理即可。

BinLake部署拓扑

在BinLake服务实际部署时，其部署拓扑说明如下。

◎一台Tower服务器：用于用户元数据、过滤规则、应用和订阅信息管理。

◎2*N*+1台ZooKeeper服务器：用于构建一个ZooKeeper集群，从而进行Wave集群管理和消息通知等。

◎一台Judge服务器：用户采集负载信息，并提供负载均衡建议决策。其中负载信息的采集是通过部署在各个Wave服务器上的Judge-Agent进程定期推送给Judge服务的。

◎*N*台Wave服务器：构成Wave集群。每台Wave服务器上部署如下2种服务。

○Wave服务：用于数据库binary log的采集并分发给下游MQ集群（Kafka或者JMQ）。

○Judge-Agent服务：用于定期采集Wave服务器的系统及Wave服务的负载和监控信息，并调用Judge服务提供的Restful接口，推送给Judge服务。

◎*N*台已经存在的线上MySQL服务器：不属于BinLake提供的服务器，而是已经存在的MySQL服务器，作为BinLake的数据源。

◎*N*台已经存在的MQ服务器：不属于BinLake提供的服务器，是已经存在的MQ服务器，处于Wave服务的下游，Wave服务会将采集到的Bianry Log Events分发给MQ集群中的Topic。

Wave服务实现原理

Wave服务的实现原理有如下3个。

1．instance的高可用。

BinLake采用ZooKeeper来实现instance的高可用。每个Wave服务在ZooKeeper上都会对应一个节点，整体流程如下。

（1）一个新的Topic产生后，Tower将会创建一个新的Topic节点。

（2）所有的Wave服务都会在新的Topic节点下创建对应的节点，ZooKeeper将会根据LoadBalance决策从这些节点中选取一个节点作为主节点，然后在主节点对应的Wave上创建instance为Topic提供服务。

（3）当提供服务的主节点对应的Wave宕机或不可用时，该Wave与节点的连接将被断开，然后该节点会被从可用的节点信息中移除，移除后，ZooKeeper集群会选出一个新的可用的节点成为主，其对应的Wave会创建instance为该Topic提供服务，从而实现高可用。

2．高效采集BinaryLog。

采用java nio高性能网络模型来实现高效采集BinaryLog，工作过程如下。

（1）Binlake集群中的每个Wave服务实例都可以与任意的MySQL实例（数据源MySQL）创建一个socket长连接，而每个socket长连接都会由Wave服务进程中的一个监听线程——这里暂称为WatchThread（Wave服务中有一个与CPU个数相符的线程池，该线程就从该线程池中取出）——进行持续监听。

（2）任何一个与MySQL实例相连的长连接产生EventLog，WatchThread都会得到通知（异步notify机制），然后将该EventLog放入队列中。

（3）线程池中的Worker Thread从队列中取出EventLog进行处理（将EventLog格式化为protobuffer格式，然后将格式化之后的消息放入Wave服务端的buffer）。

（4）buffer size打满之后，批量发往MQ的消息队列中。

各个线程之间的工作互不影响，不会因为转换导致socket读事件阻塞，整个设计采用生产消费模型。

3．分发Binlog Event到Topic。

Binlog Event采用批量发送的方式发往消息队列MQ，消除发送端性能瓶颈，流程如下。

（1）从转换器converter获取到一条消息，放入buffer。

（2）重复步骤（1），直到buffer的size达到指定阈值，然后执行步骤（3）。

（3）将buffer中的消息批量发往MQ消息队列，并重复步骤（1）和步骤（2）。

批量发送过程采用异步notify机制，不会出现因同步网络发送而引起的CPU使用率过高的问题。

Tower服务实现原理

Tower服务的实现原理是instance复用及日志过滤，如图2-3所示。其具体说明如下。

◎Wave中的每个instance与实际的数据源MySQL实例一一对应，针对同一个MySQL实例，无论多少个业务方采集，都只会由一个instance进行采集。

◎每个业务方都可以针对一个MySQL实例中的数据设置filter，从而只过滤出自己感兴趣的EventLog，也可以查阅某个instance对应的所有的过滤规则，并复用过滤规则。

◎Wave中的instance对采集到的Eventlog，使用设置的过滤规则依次进行过滤，然后发送到指定的一个或者多个Topic中。

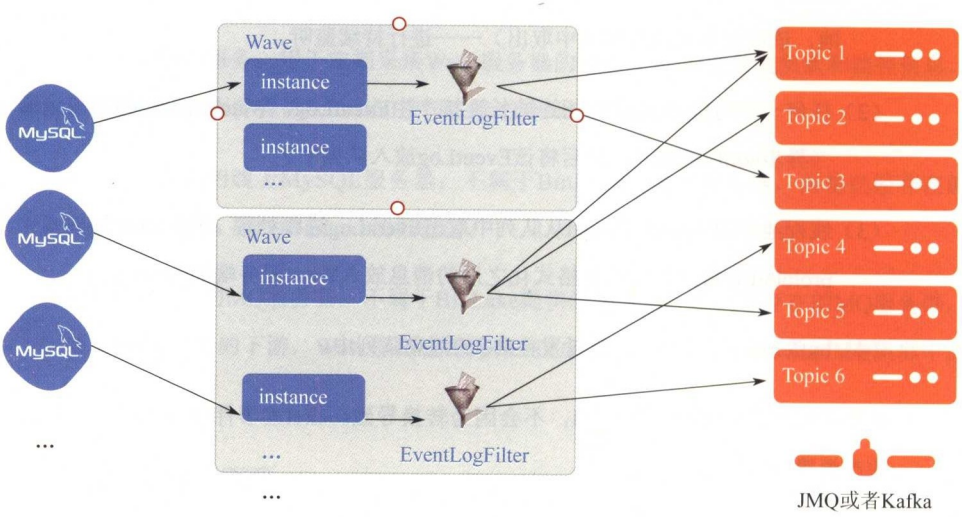


图2-3　BinLake instance复用及过滤规则

当前限制

当前架构版本限制如下。

◎目前版本只能采集和订阅MySQL的Binary Log。

◎MySQL的Binary Log的Format只能是Row Base的。

◎MySQL中的表必须要有主键。

2.3 弹性数据库

弹性数据库（Jingdong Elastic Database，JED）是京东商城十年数据库生产经验的总结与升华，兼容MySQL协议，是适合海量数据事务处理、分析计算、动态扩展、灵活复制、自动备份恢复、自动历史结转、日志订阅、完全容器化部署的分布式数据库中间件。其特点具体如下。

◎在线、透明的弹性扩展能力：纵向扩容——基于底层JDOS容器平台实现透明的scale up；横向扩展——动态重新分片（Resharding）、跨IDC部署，实现服务器与数据中心两级可扩展性。

◎完全兼容MySQL协议：兼容现有应用系统，利于系统升级、迁移。

◎整体解决方案：集成数据库日志订阅及消费、自动备份恢复与历史结转。

◎高可用保证：所有节点无单点故障，以及精确、及时的故障检测与自动恢复。

◎数据可靠性：默认三副本存储、跨两个IDC，其中同IDC主从副本semi-sync复制、异地IDC副本异步复制。

◎完全容器化部署，运行在容器平台JDOS之上。

2.3.1 JED整体架构

JED的整体架构如图2-4所示，其核心设计思想是采用分布式存储的技术路线，通过两级抽象，来有效地解决数据库的可扩展性问题。

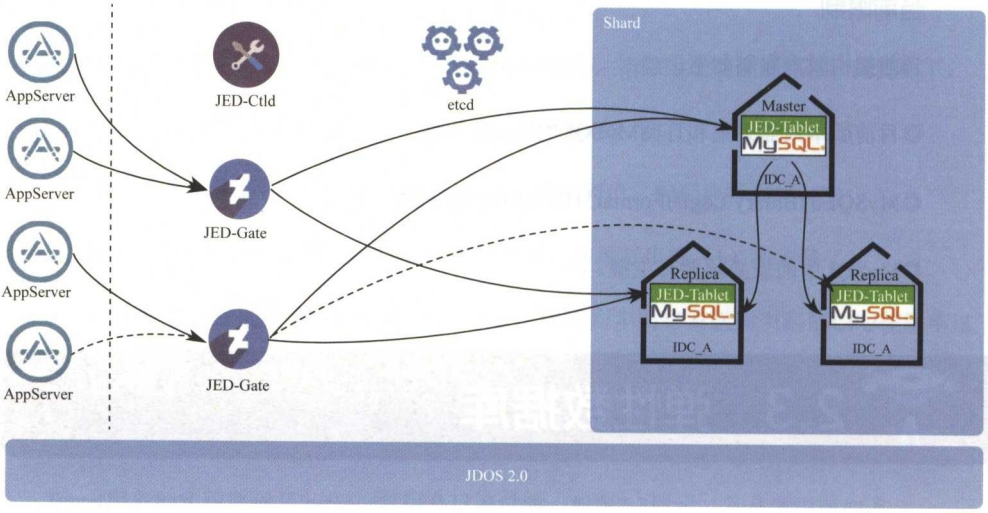


图2-4 弹性数据库整体架构

架构主要模块如下。

◎元数据管理服务(Topology)。Topology负责JED元数据信息存储服务，即存储数据库分片方案、主从状态信息等重要元数据。Topology是基于一致性存储方案（如etcd）实现的。用户可以通过使用jedctl（命令行）和jedctld（Web）访问Topology数据。

◎数据节点。每个数据节点部署在一个Pod上，该Pod包含两个Container，分别提供JED-Tablet服务和MySQL数据库服务。Replica和ReadOnly类型数据节点上的MySQL服务是Master类型数据节点上的MySQL服务的从库。JED默认在每个Shard中有三个数据节点，这三个数据节点的类型分别为：Master、Replica和ReadOnly。

◎接入代理（JED Gate）。Gate是一个轻量级的代理服务器，兼容标准的MySQL协议。其接收客户端连接，将SQL请求路由到正确的Tablet，并且将返回结果合并后再返回给客户端。Gate缓存Topology服务中的元数据信息，且接收变更通知来实时同步元数据更新。通常每个机房部署一组Gate节点，服务该机房所有应用。

2.3.2 聚合查询

在分布式数据库JED中，所有的表都需要设置一个拆分字段，比如user表，我们设置user_id为拆分字段。客户端通过连接Gate对数据库进行增删改查操作。客户端连接到Gate以后，将需要执行的SQL发送到Gate，Gate对SQL进行解析，根据解析结果、路由字段等信息，将SQL

发送到相应的Shard，待每个Shard返回执行结果后，Gate将所有的结果进行汇总，将汇总后的结果返回给客户端。

比如，客户端发送“SELECT * user WHERE user_id = 1;”，JED就会根据user_id的值和路由信息，解析出user_id为1的数据存储到了Shard0，然后将SQL发送到Shard0，然后根据Shard0返回的结果返回给客户端。

上面是涉及一个Shard的情况，如果涉及多个Shard呢？

比如“SELECT COUNT(*) FROM user;”，Gate后面有两个Shard，Gate将这条SQL发送给两个Shard，Shard0返回100，Shard1返回50，Gate将两者求和，返回150给客户端。

SUM、MIN、MAX和COUNT的聚合类似。

但是，AVG和聚合却不能直接将SQL发送到后端Shard，比如“SELECT AVG(AGE) FROM USER;”。

Shard0上有100条用户数据，平均年龄是20，Shard1有50条用户数据，平均年龄是30。如果按照之前计算COUNT的做法，直接计算(20 + 30)/2=25是错误的。我们应该计算（100 * 20 + 50 * 30）/150≈23.33。

所以，这种情况下我们就需要对发送到后端的SQL进行改写。将每个Shard的人数计算出来发送到Gate作为权重进行汇总。

这时候，可以这样改写SQL：“SELECT AVG(AGE), COUNT(*) FROM USER;”。

Shard0返回{20.0,100}，Shard1返回{30.0,50}，这时我们就可以计算出平均年龄了。

除了上面提到的改写方式，是不是也可以改写成“SELECT SUM(AGE), COUNT(*) FROM USER;”？思考一下，两种改写方式各有什么优缺点呢？

如果SQL中带LIMIT，就需要在Gate中进行筛选。比如“SELECT * FROM USER LIMIT 10;”。

这种情况下，不对SQL进行改写，直接发送到后端，后端返回结果取前10条数据返回给客户端。但是，如果SQL为“SELECT * FROM user ORDER BY AGE LIMIT 10;”，就需要用Gate进行排序了。

下面再看一下Gate是如何做排序的。为了避免业务大数据量的查询，Gate支持流式数据处理。

对于查询操作，MySQL提供了mysql_use_result()和mysql_store_result，调用mysql_store_result将从MySQL服务器一次性查询所有的数据到客户端，然后读取。调用mysql_use_result并不会一次性从MySQL服务器读取数据，需要一直循环调用mysql_fetch_row分批读取数据，直到返回NULL。在查询数据量比较大时，比如大数据抽数操作，一次读取大量的数据到内存很容易撑爆服务器内存，所以，如果查询数据量比较大，我们就要分批次读取数据，然后分批次返回给客户端。

如果涉及排序操作，则将排序下推到MySQL，这时每个节点返回的结果已经有序，虽然每个节点只返回了部分数据，但是可以认为这是k路归并问题，并且每一路数据已经有序。所以，这时我们把k路的第一个数据取出来比较，得到一个最小的数据，就是所有节点中最小的数据，这条数据就可以发送到客户端了。

一边从后端读取数据，一边排序，一边把排好序的数据发送到客户端，这就实现了流式的排序。并且我们设置了双层buffer，buffer是固定长度的阻塞队列，需要排序的k路数据，将其放到对应的srcbuf中。如果srcbuf满了，就不会再去后端读取新的数据。将排好序的数据放到sortedbuf，用专门的线程去sortedbuf中读取排好序的数据并发送到客户端。如果排序速度快，往客户端发送数据慢，就会导致sortedbuf变满，向sortedbuf推数据就会阻塞。同样的道理，如果排序比较慢或者往前端发送数据慢，各个srcbuf满了，这时读取后端数据的线程就阻塞在了向srcbuf推数据的操作上。固定大小的srcbuf既避免了内存暴涨，同时也兼顾了排序效率。各个Shard返回的数据已经是有序的了，直接做归并即可。如果Shard过多，每归并一行数据就需要遍历所有的srcbuf第一行数据，也相当耗费资源。为了避免这样的情况，JED把所有的srcbuf组织成一个优先队列，按照srcbuf队列头元素作为key定义优先级，每次从优先队列pop出一个srcbuf，从srcbuf中读取一行数据，然后将srcbuf再次push到优先队列中，如果从srcbuf读到EOF，则不再push。

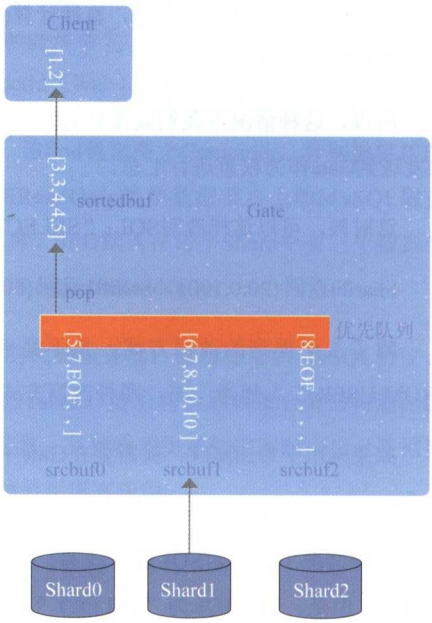


图2-5　流式排序示意图

2.3.3 弹性扩展能力

有效地支持动态重新分片以获得横向扩展性，是JED的核心功能。

主要概念

1. Keyspace。

   Keyspace是逻辑上的数据库。

   在非拆分场景下，一个Keyspace 直接定位到一个指定的数据库。对Keyspace的读写操作和直接对相应数据库进行读写操作类似。

   在Shard场景下，一个Keyspace会对应多个MySQL Database，这时对Keyspace的读写操作会由jedgate根据路由配置信息路由到一个或者多个MySQL Database上。

2. Shard。

   一个Shard就是在一个Keyspace中的一个水平分区或者分片。每个 Shard 都会包含一个Master实例和多个Slave实例。而且不同的Shard之间不会存在数据重叠的现象。

3. Resharding。

   JED支持动态Resharding，也就是说JED的Resharding过程可以把一个或者多个Shard分割成更多更小的Shard，也可以将相邻的Shard合并成一个Shard。

4. Key Range。

   JED使用 Key Ranges来决定应该由哪个Shard来处理特定的读写操作。

   Key Range是连续的Sharding Key序列。一个Key Range有起始值和终止值，如果一个Key大于等于某个Key Range的起始值且小于终止值，那么这个Key就落到了这个Key Range中。

   一个空的终止值代表最大值，一个空的起始值代表最小值。

   JED在路由计算之前先将Sharding Key转换成字节数组，如果整个Range是从[0x00]到[0xFF]的，那么[0x80]则是Sharding Key的一个中间值。比如，一个Keyspace如果有两个Shard， Sharding key对应的字节数组小于[0x80]的可以落到一个Shard上，大于或等于[0x80]的则落到另一个Shard上。

   用如图2-6所示的线段来代表Key Range，线段的最小值为[0x00]、最大值为[0xFF]。从[0x00]到[0xFF]的整条线段代表了整个Key Range。

Start=[0x00], End=[0xFF]: 整个Key Range。

Start=[0x00], End=[0x80]:小于0x80的 Key Range，整个Range的前1/2。

Start=[0x80], End=[0xFF]:大于0x80的 Key Range，整个Range的后1/2。

Start=[0x40], End=[0x80]: 第二个1/4 Key Range，整个Range的1/4。

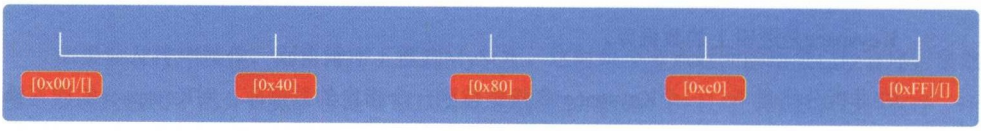


图2-6　Key Range

Resharding步骤

第一步：启动两个新的Shard

我们以将一个Shard Resharding成两个Shard为例，来介绍Resharding的过程。起初只有一个Shard，Key Range为Start=[0x00], End=[0xFF]。

后来业务发展过快，单机数据库实例难以满足业务需求，需要做Sharding，将数据分到两个Shard上（Shard架构如图2-7所示），Key基本服从均匀分布，所以将Key Range分别设置为如下的形式：

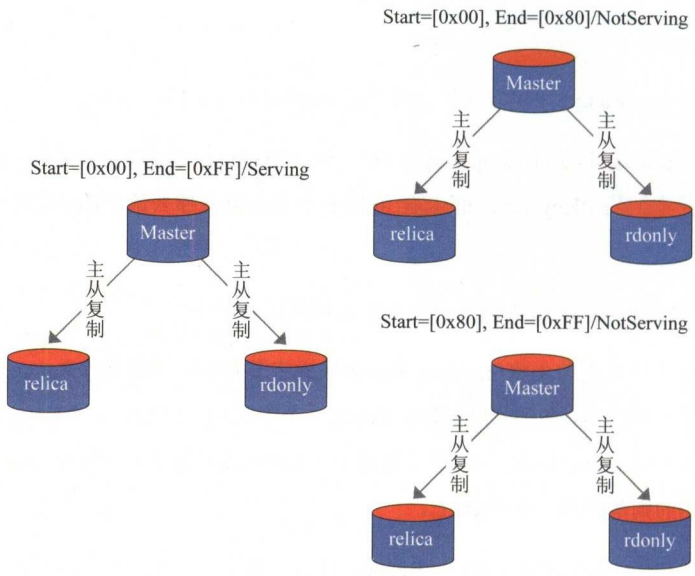


图2-7　Shard架构

Shard0:Start=[0x00], End=[0x80]

Shard1: Start=[0x80], End=[0xFF]

第二步：复制表结构

初始化Destination Shard后，下一步将Source Shard的表结构信息复制到Destination Shard上。

第三步：迁移数据

1. 初始化Source Shard、Destination Shard信息。

迁移数据过程中，JED会自动去寻找所有的Shard并根据Key Range将所有Shard分成Range没有重合的两份，两份分别标记为Left、Right，然后通过查询Shard是否在服务中（Serving/NotServing）来区分Source Shard和Destination Shard。一般而言，处于Serving状态的是Source Shard，而处于NotServing状态的是Destination Shard，如图2-8所示。

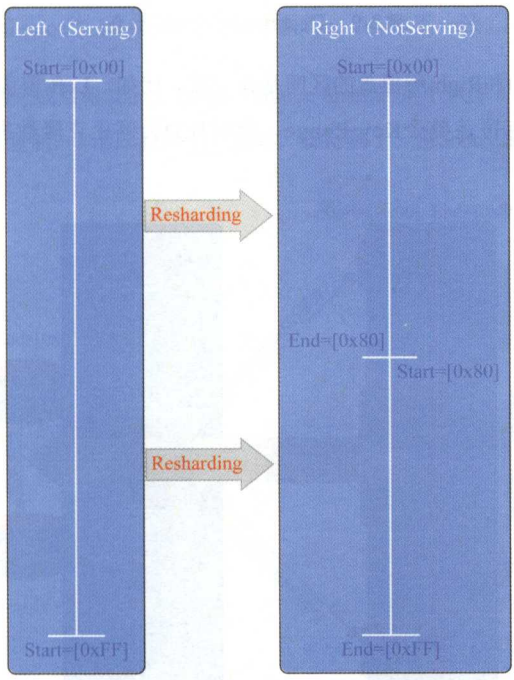


图2-8 Resharding

需要注意的是，JED不支持多于两个Shard同时覆盖到任意一部分或者一段Key Range的情况。比如，在同一个Keyspace里有[0x00]～[0x80]、[0x00]～[0x60] 和

[0x40]～[0x80]三个Shard，[0x40]～[0x60]这一区间被三个Shard覆盖，这时JED不确定将数据复制到哪个Shard。JED就会停止克隆数据并提示错误信息，如图2-9所示。

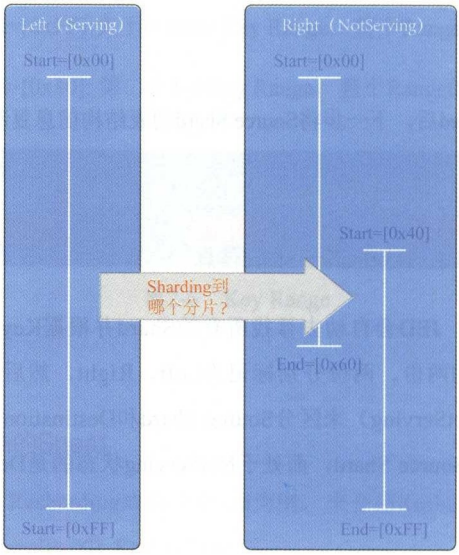


图2-9　Destination Shard重叠

JED会检查Left和Right所覆盖的区间是否一致。比如，Left 覆盖的是整个Key Range，但是Right没有覆盖整个Key Range，此时JED会停止克隆数据并提示错误信息，如图2-10所示。

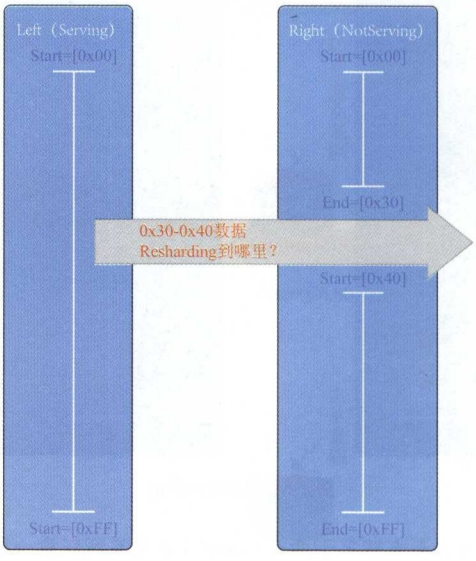


图2-10　Left、Right覆盖区间不一致

2．克隆数据。

（1）选择Source Shard的一个从库，我们称之为Source Tablet，在每一个Destination Shard中选择一个主库，我们称之为Destination Tablet。

（2）复制数据。JED给每个Destination Tablet创建一个insertChannel，每个insertChannel都有插入缓冲和流量控制器Throttler（可根据网络流量、机器内存、CPU使用率、复制延迟等多个参数进行流量控制）。

insertChannel类似一个队列，有一生产者去Source Tablet批量读取数据，并根据Sharding Key将读取到的数据分别插入到与Destination Shard相对应的insertChannel上，有消费者从insertChannel取出数据，然后将读取的数据直接插入到对应的Destination Shard。如果Destination Shard有脏数据，那么在克隆数据的过程中，JED会将脏数据删除或者修正；反之，在克隆数据之前将数据插入到Destination Shard是不可以的，数据会被删除！

需要注意的是，克隆数据过程中，Source Tablet的数据库可能会有更新，所以克隆到Destination Tablet的数据可能和Source Tablet不一致。

（3）复制数据完成后将Source Tablet（Source Shard的某个从库）从MySQL主库上摘下，停止主从复制，如图2-11所示。这时，就得到了一份Source Shard的快照数据。

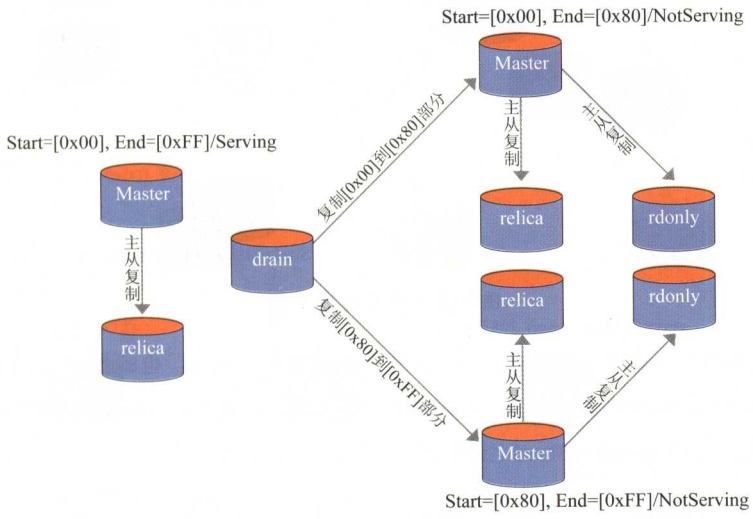


图2-11 克隆数据

（4）重复步骤（2）。因为现在Source Tablet中存有的是一份快照数据，不再进行更

新，所以执行完成后，Source Tablet和Destination Tablet的数据就是一致的。因为之前执行了步骤（2），Destination Shard中已有数据，所以步骤（4）执行速度比步骤（2）快很多。这样做的目的也是为了避免Source Tablet停止服务时间过长，因为Source Tablet还需要对外提供复杂查询、数据分析等服务。步骤（2）是可选项目，通过参数控制，跳过步骤（2），JED仍然能正确地克隆数据，但是Source Tablet会停止服务时间过长。

（5）已有数据克隆完成后，将Source Tablet重新挂到主库下开启主从复制。

（6）在Destination Tablet上开启过滤复制，过滤复制就是在Destination Tablet上开启一个BinlogPlayer，BinlogPlayer去Source Tablet读取Binlog，根据Binlog和Sharding Key决定是否执行Binlog，开始过滤复制后，Destination Tablets就能追齐因步骤（4）落下的数据，并保持Destination Tablet和Source Tablet数据的一致（忽略复制延迟），如图2-12所示。

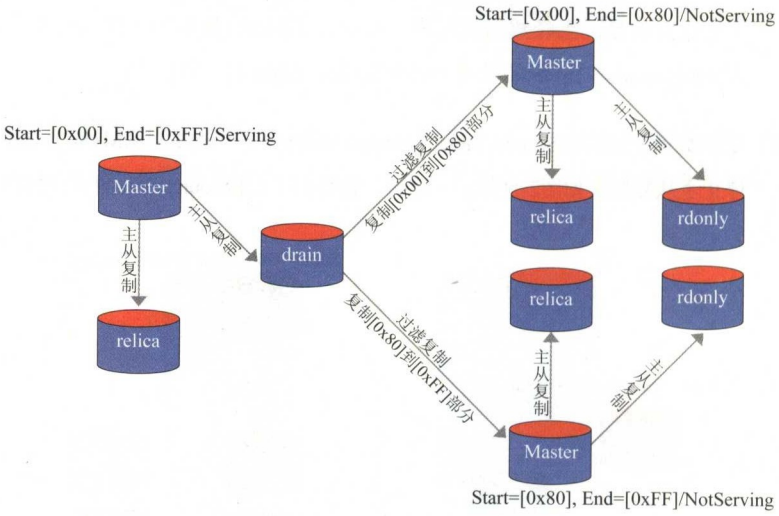


图2-12　开启过滤复制

需要注意的是，克隆工作完成后，过滤复制仍然是开启状态，也就是说，忽略复制延迟的情况下，Source Shard和Destination Shard的数据是一致的。现在，Source Shard的任何更新操作都可以在相应的Destination Shard上读取到。

第四步：数据一致性校验

JED做数据一致性校验的原理和克隆数据类似，将Source Tablet从MySQL主库摘下，通过

JED自身的过滤复制保证Source Tablet和Destination Tablet没有复制延迟。这时再对表里的数据逐行进行比较，比较完成后再将Source Tablet挂到MySQL主库下。

第五步：切换到新的Shard

若数据一致性校验通过，则可以切换到新的Shard上。迁移过程中首先需要迁移从库，最后迁移主库。因为如果首先迁移了从库，从库还是原来的Shard的从库提供服务，这时原来Shard中从库的数据已经不更新了，那么就相当于从库在使用一份历史快照数据提供服务。

下面介绍迁移主库的主要步骤。

1. 停止Master的读写服务并得到Master的Binlog位置pos。
2. 等待所有的Destination Shard追Binlog到 pos位置，保证无复制延迟。
3. 停止所有Destination Shard的过滤复制。
4. 修改Destination Shard的servedType为服务状态，更新路由信息并广播。
5. 最后确认服务正常，停止Source Shard的所有Tablet。

迁移从库的过程类似，便不再做介绍。

整个迁移过程完成后，原来的Shard不再提供服务，将所有的读写操作路由到新添加的两个Shard上，如图2-13所示。

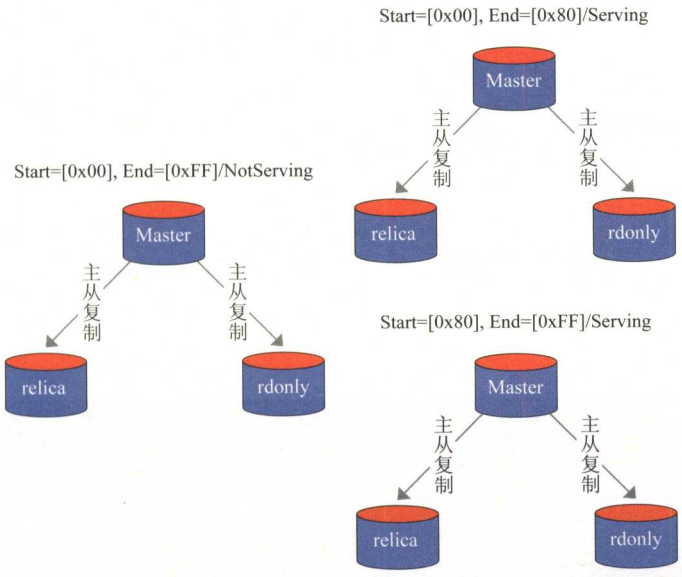


图2-13 切换到新的Shard

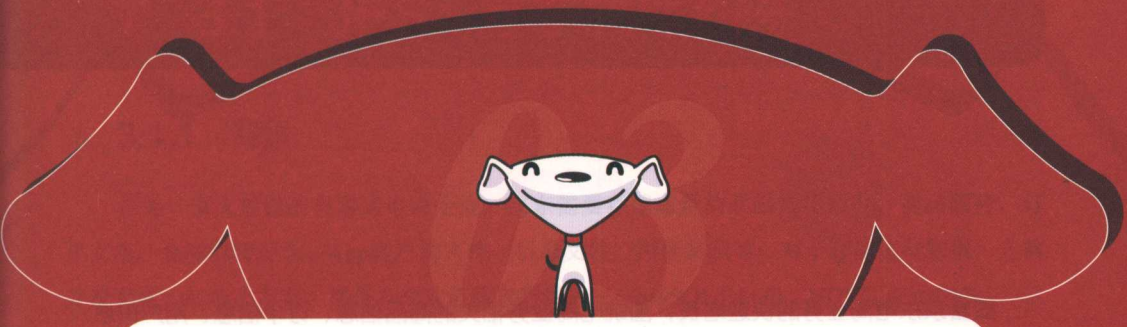
Resharding过程对业务的影响

以上就是JED做Resharding的主要过程，从中可以看出，对线上业务产生影响的只是“切换到新的Shard”这一步。在切换到新的Shard的过程中需要先停止Master服务，Destination Shard没有复制延迟后停止过滤复制、更新路由。线上没有很长的复制延迟时，整个切换过程几秒钟就可以完成。

垂直分片

除了上文中提到的水平拆分，JED还支持垂直拆分。垂直拆分的意思就是根据业务的相关性将表放到多个Keyspace中。比如一个很复杂的订单系统中涉及了用户信息、商品信息，所有的数据都存放到了order_keyspace中。如果做垂直拆分，可以将用户信息放到user_keyspace中，商品信息放到product_keyspace中。垂直拆分实际是做了业务拆分。同样，整个垂直拆分过程对读操作没有任何影响，仅有几秒钟的时间不能写入数据。垂直拆分的数据迁移思路和水平拆分类似，不再做详细介绍。

JED作为京东商城自主的弹性数据库中间件，在2017年5月投入生产，成为线上业务默认的数据库服务（Database as a Service）。特别地，其跨IDC部署及管理的能力使得JED成为商城异地多活架构升级项目的重要组成部分，这将在第5章谈到。



第3章
分布式存储技术

3.1 JFS：京东文件系统

3.2 JIMDB：内存是新的磁盘

3.3 FBase：大表存储

3.4 Container File System

随着京东电商业务的快速增长，围绕电商业务相关的商品信息、订单信息、用户信息、仓储信息也在迅速膨胀，这些信息的高效、安全存储成为了一个很大的挑战。我们有PB级别的图片、订单快照等文件需要存放，也有单个业务几十亿条记录的表需要被快速存储和访问，TP99要小于10ms，传统的文件系统、关系型数据库完全满足不了业务的发展需求。

开源的产品都无法直接满足需求，针对不同的产品，我们有的采用完全自研的方式，有的在已有的开源产品上进行改造和创新。近几年，根据京东业务自身的发展特点，我们开发了用于存放大文件、小文件的对象存储系统，持久化与非持久化的K-V存储系统，以及应对海量数据的宽表存储系统。

京东每年都会进行618和双11大促，大促时访问的洪峰经过业务系统的层层调用，最终都会转化为对存储系统的访问，网页丰富的商品信息展示、用户个性化的搜索结果和商品推荐，背后都需要实时地从存储系统中读取数据进行计算，用户打开的每一个页面都会转化为从多个系统中访问数据，存储系统的稳定和性能直接决定了用户购物的体验。

在开发这些存储系统的过程中，面临了很多的挑战。怎么精确、及时地发现故障，自动进行故障恢复？怎么快速地进行横向扩容、数据迁移？怎么解决系统的读写性能？每一个问题被解决的背后都有着开发人员夜以继日的奋斗。本章将会详细介绍京东存储系统的建设历程和架构。

3.1　JFS：京东文件系统

3.1.1　背景

作为一家大规模的自营式电商企业，京东需要存储海量的非结构化数据：商品图片、订单文本、仓库流转记录、App客户端文件、日志文件、内部文档等。对于存储这些数据，之前并没有统一的解决方案，都是各个业务线自行解决——MySQL BLOB、HDFS、FastDFS。

2013年5月，京东开始组建存储组，自主研发JFS——京东文件系统，以实现非结构化数据存储统一服务为目标。

3.1.2　小文件存储

针对3个典型的应用场景——商品图片、OFC订单、WMS库房流水，JFS第一版定位为海量小文件存储，其核心功能定义如下。

◎海量小文件存储，极高的可靠性、可用性与一致性。

◎Key-File数据模型，Key由系统生成，全局唯一；文件immutable，即不可修改，甚至极少被删除。

其主要包含如下3个模块。

◎ZooKeeper作为集群协调器管理元数据信息。

◎由Go语言开发的DataNode，实现服务端读写逻辑、复制协议、故障恢复等。每个DataNode管理一块磁盘——该设计大幅简化了工程实现。

◎由Java开发的客户端。

复制协议实现了一种Paxos变体，或者说一种极简的Paxos实现，如图3-1所示：固定成员（一个复制组由1primary + 2follower构成）、固定角色（primary与follower角色不会发生变更）、固定读写流程（client将写操作发送到primary，它在写本地的同时将写操作发给两个follower，三副本都写入成功后才成功返回给用户；优先在follower上读取，提高系统的并发能力）。

存储引擎采用Append-Only方式，每个DataNode维护一组（默认配置为512）Chunk大文件，客户端上传的小文件（如一张图片）被并行追加至一个复制组三名成员对应的Chunk中，如图3-2所示。

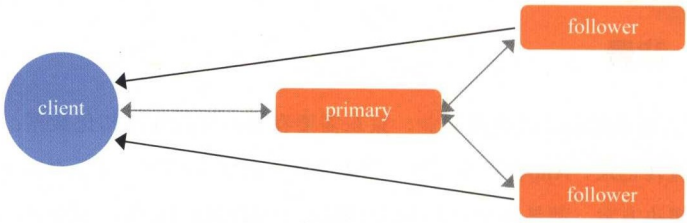


图3-1　JFS小文件复制协议

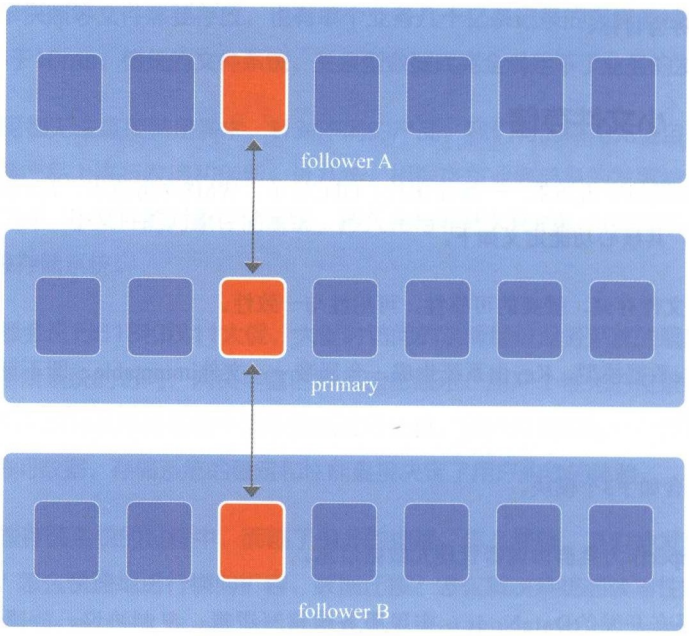


图3-2　JFS小文件数据存储

JFS为每个成功上传的小文件生成全局唯一的JFS Key来编码其存储位置信息：

JFS Key = Replica Group ID/Chunk ID/Offset/Length/Checksum/Signature。

比如，jfs/t3442/251/2127752103/150148/57583d02/5844d73fNaca4af3d.jpg 是京东网站上一款电饭煲的主图的JFS Key，表示该图片存储在3442号复制组、251号Chunk的2127752103字节偏移处，长度为150148字节，CRC校验码为57583d02，签名5844d73fNaca4af3d用于防止URL

篡改攻击。

3.1.3　图片系统

基于JFS小文件存储系统，我们在2014年春天重新建设了京东商品图片系统（系统架构如图3-3所示），并在公司上市之前成功上线。之后，图片系统零故障稳定运行至今，历经商品图片规模从十亿到百亿的大幅增长。

同一张商品图片可能有数十种不同的规格（不同的设备、展现格式、降质参数），但源站JFS只存一副原图，CDN会缓存各种规格的图片URL，CDN未命中的图片则进行回源实时处理并返回。这样不仅节约了源站JFS的存储空间，也可以灵活地满足业务不断变化的需求。

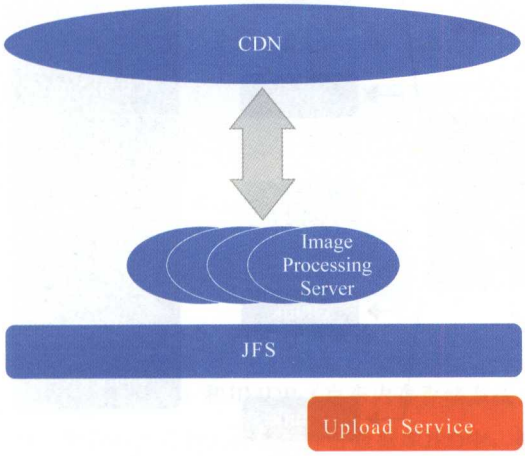


图3-3　京东新图片系统架构

在解决最核心的图片存储和处理问题后，我们也做了很多工作来推动图片技术的发展。在缩放效率上，引入ICC、IPP编译将图片缩放性能提升到最初的3倍以上。在流量优化方面，将Webp格式引入京东，与无线部门紧密合作，将移动端的图片全部替换成Webp格式，给用户节省约35%的下行流量，并显著提升了用户体验。

3.1.4　大文件存储

JFS V2实现大文件存储功能。对于大文件写操作来说，类Paxos复制协议并不合适。primary拿到数据后同时发送给两个follower，这样primary的带宽资源将成为系统的瓶颈。因

此，在大文件存储复制协议的选择上，JFS采取了链式复制（Chained Replication）以提高写操作吞吐量。链式复制结构如图3-4所示。在数据发送和接收上，也均使用了流水线处理，进一步提高了数据传输效率。

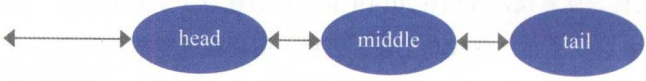


图3-4　JFS大文件复制协议

在数据存储结构设计上，恰恰与小文件相反，将一个大文件分成多个块来存储，这样可以规避局部过热的文件造成单机磁盘I/O过载；另外，分成多块也更利于整个系统资源的调度。大文件的数据存储如图3-5所示。

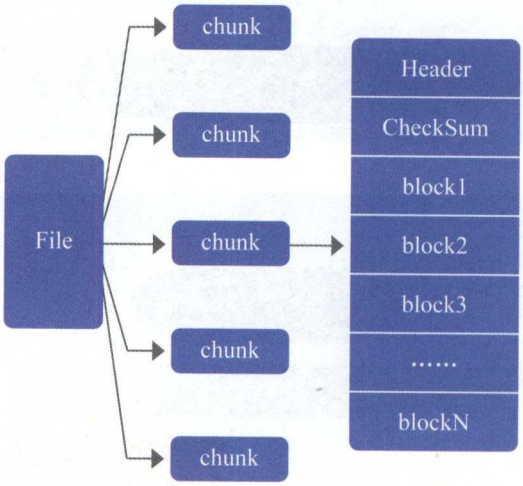


图3-5　JFS大文件数据存储

3.1.5　对象存储服务

JFS的小文件存储和大文件存储功能，从可靠性、可用性和稳定性方面，已经满足了大部分的业务需求，但使用起来却不是很方便，上传和下载都需要通过SDK，用户排查问题不是那么便捷，且对多语言的支持也不好。我们构建了JFS V3产品形态：简单对象存储，支持HTTP协议；支持文本、图片、视频等任何类型数据的存储；支持1个字节到1TB大小的数据存储；支持List操作，用户数据可以有层次结构。JFS V3为众多业务场景提供了最便捷的数据访问方式。

对象存储系统架构如图3-6所示。除了前面已经提到的大小文件存储，还需要构建Gateway、账户和Bucket管理、日志处理等，当然还有最复杂的元数据管理。

对象存储的元数据管理是一个业内难题。虽然对象存储并无目录的概念，但要支持按前缀进行List的操作，即能通过Prefix和Delimiter的结合，实现层次查询，是有一定难度的。在数据量不大时，类似于Hdfs的NameNode将全部用户Key都存在内存中就能满足需求，但当对象的数量超过十亿时，将会耗尽内存，无法做到横向扩展。很多KV存储能做到随意横向扩展，却不能很好地支持对象存储List请求。

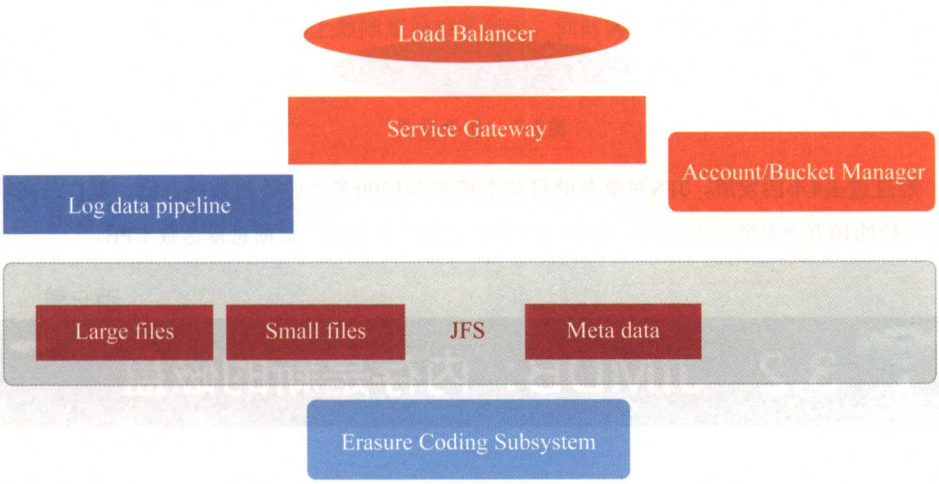


图3-6　对象存储架构

JFS V3采用JED（京东弹性数据库）和JIMDB（京东内存存储系统）组合，来实现对象存储元数据的有效管理。将元数据扁平化持久存储在弹性数据库JED中，热点缓存在JIMDB中，一方面利用JED的单库MySQL的B树结构实现元数据的List层次查询；另一方面使用JIMDB实现高速单Key查询。当数据量达到一定阈值时，JED可以进行在线的扩容与重新分片，JIMDB也可以做动态容量扩展，这使得JFS V3服务逻辑层在工程实现上非常简单。

电子签收

JFS V3作为对象存储服务，一经推出就受到业务部门的广泛欢迎，电子签收小票的存储管理就是一个特别典型的应用。

从环保和成本的角度出发，运营系统青龙研发部创新性地启动了电子签收项目，取代之前每天数百万张的纸质小票。电子签收产生的海量签名图片需要高安全性、高稳定性、高持

久性地保存。这无疑是对象存储的一个很好的应用场景，在此之上我们还实现了加解密、文字转图片、图片合成等定制化需求。基于JFS对象存储的电子签收后台系统，根据传回来的签收信息，按照指定样式生成签收小票图片并与用户签名图片合成，再按照业务安全性要求做数据加解密处理。如图3-7所示。

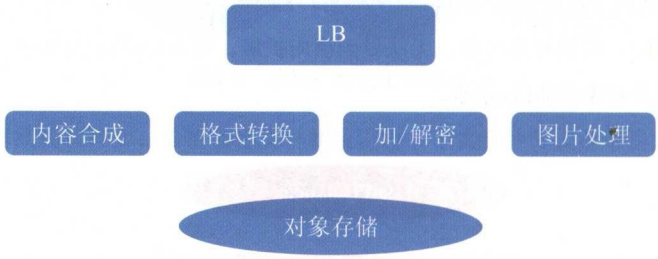


图3-7　电子签收

经过过去4年的发展，JFS对象存储目前支持京东1200多个业务的数据存储，双11最高峰值为每秒约10万个对象同时读写，存储对象数目达数百亿级别，数据总量达数十PB。

3.2　JIMDB：内存是新的磁盘

3.2.1　缓存的大背景

缓存在软件应用特别是在互联网应用中无处不在。从数据库到应用服务，再到前端的页面，每一层都会使用缓存进行加速，即使是硬件产品（比如CPU、磁盘、网卡等）也都会有相应的缓存或缓冲区。

当一个网页被打开时，为了提供良好的用户体验，提高用户购买的转化率，一个纯静态的页面往往无法满足业务的需要，后台会有几十个、上百个服务为这个页面提供动态的个性化数据。比如，根据用户过往的购买记录和上网的浏览信息帮他推荐感兴趣的商品，告诉用户这些商品购买比例如何，好评度怎么样，什么时间段可以送货到家，这个商品有没有促销，能不能用券，如果缺货需要提醒用户这个商品当前是预定状态，还有很多就不一一列举。这么多的服务需要调用，而且要在每秒成千上万次请求的情况下，毫秒级将结果展示在用户的显示屏中，除了良好的架构方案，缓存的极致使用也是必不可少的。

有些业务团队在前期使用memcached服务，固定部署几个实例进程，客户端通过Hash算法把数据切分成几份，由于业务量相对小一些，所以能应付当时的业务场景。服务端也没有做主从热备或者自动故障恢复等方案，服务宕机就让运维重启一下。当数据内容增多了需要扩容时，就找一个业务访问量低的时间段，暂停一小会服务把数据整体重新分布一下，有的小组会写一个迁移工具，有的直接从数据库加载数据重新推送到memcached中。后来，Redis慢慢流行起来，支持更丰富的数据结构，业务使用起来更加方便。但是在用法上并没有什么改变。大家还是按照以前的思路，在客户端做一些封装，服务端地址直接写死在客户端应用的配置文件中，服务端地址改变了就重启客户端程序。

缓存的规模越来越大，线上故障变得频繁起来，同时客户端实例数也在增加。业务需要一个强有力的缓存平台：能够自动进行故障恢复、在线扩容、负载均衡，而且这些过程要完全在应用无感知的情况下完成。

3.2.2　JIMDB登场

第一版

JIMDB项目在2014年年初立项启动，基于Redis而又兼容Redis的缓存或者叫内存存储平台，主要解决几个核心问题：精确的故障检测和自动故障切换，在线无损扩容，完善的监控和报警服务。

故障检测和切换

在故障检测和故障切换的方案中，比较容易想到的方案就是引入ZooKeeper，通过ZooKeeper的临时节点探测不存活的服务，但是由于需要修改服务端代码、不便于跨机房部署、watch数目和连接数过多有性能问题等原因，最终没有被采用，而是自己开发了一套分布式探测程序。这个探测程序主要用来检测JIMDB实例的存活状态，但是它必须尽可能地避免部分网络不通时导致误判的问题。采用的方案就是，部署多个探测程序实例到机房的不同机架里（如图3-8所示）。多个探测实例对同一个JIMDB实例进行探测并进行分布式投票（distributed voting），只要一个探测实例检测到服务端实例是存活的，那么该实例就被认为是存活状态；当没有人反馈其为存活状态，且超过半数的探测实例认为该实例死亡时，则判定该实例故障，通知故障恢复程序进行主从切换，变更集群拓扑结构，并把新的拓扑结构通知给所有的客户端。故障检测和恢复的问题算是基本解决了，接下来需要解决扩容的问题。

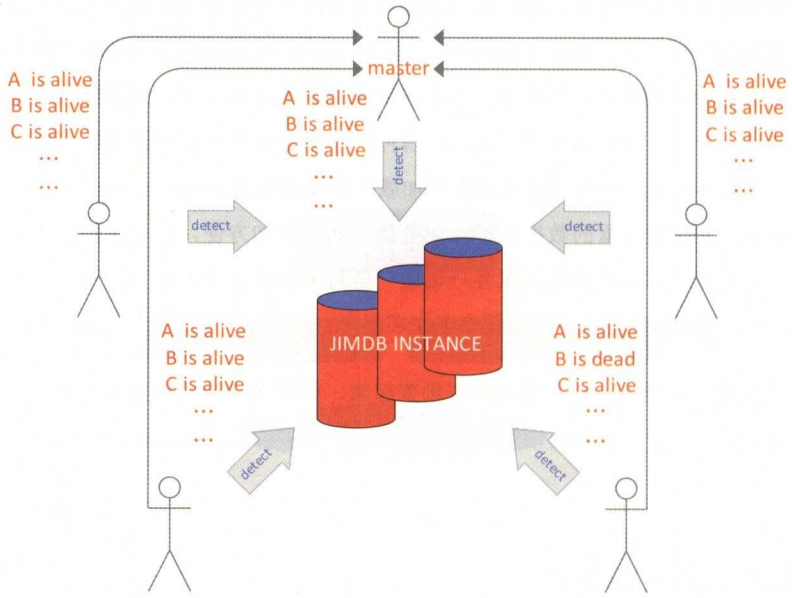


图3-8 JIMDB探测方案

无损扩容

为了满足扩容的需求，客户端采用了一致性Hash算法，预先在客户端对数据划分了一个比较大的slot集合，一段连续的slot对应一个Shard，每个Shard由一主一从或者一主多从组成，当主发生故障时可以让从提升为主，继续提供服务。

业务在刚上线阶段，访问量和需要缓存的数据量并不大，提前申请很多的缓存服务会导致资源的浪费，随着业务的增长及促销活动的开展，访问量和数据量变大，缓存服务端需要扩容。按照一致性Hash算法，需要让原来落在同一个Shard上的一段slot区间进行分裂，变成两段区间，每段再各自对应一个Shard。这意味着这个Shard上的数据有一部分需要迁移到新的实例上完成扩容。这些slot的信息在服务端并不存在，因此服务端也不知道哪些数据应该进行迁移，在扩容过程中还需要保证客户端能够正常访问。解决方案是采用代理的方式（如图3-9所示），当需要对某个Shard进行扩容时，先下发一个新的拓扑信息给客户端，让访问该Shard的请求全部变更为访问代理服务，代理服务再把请求转发给该Shard。当所有客户端都连接上代理程序后，扩容就可以开始了，数据同步程序通过复制协议从原来的实例上复制数据，再过滤出需要迁移的数据，并转发到新的实例上。当代理程序检测到数据复制快要完成后会挂起所有的请求，等待复制完成，然后按照新的slot分布信息把迁移的Key请求转发到新的服务器上，这样扩容就完成了最主要的步骤。接下来，只要把分裂后的拓扑信息下发给客户端，客

户端便会重新连接服务端，把与代理端的连接断开，扩容就算完成了。

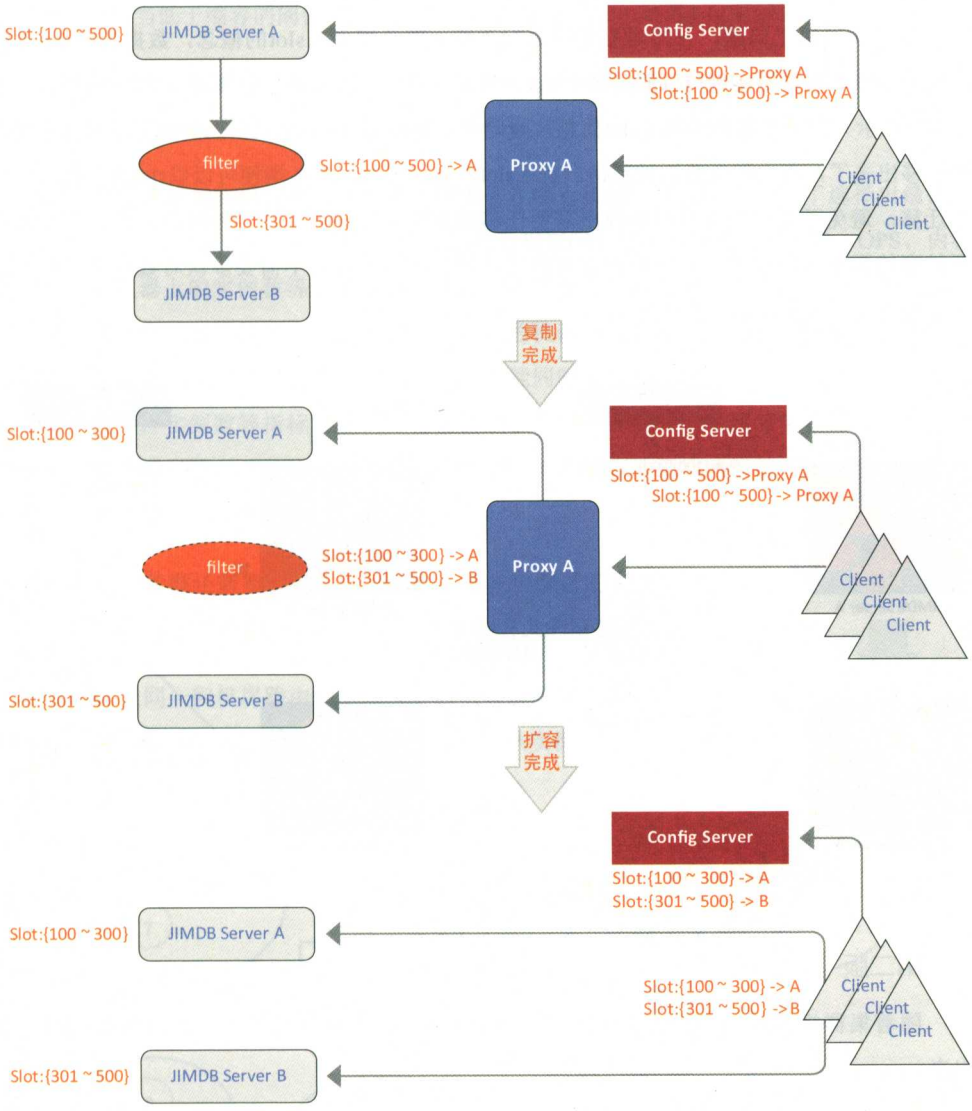


图3-9　基于代理层扩容

虽然上面的流程能够完成扩容过程，但是在扩容时需要从服务端拉取全部的数据进行过滤，然后再转发；当数据量比较大时，在当时的千兆网卡下，为了不影响正常业务，需要控制数据过滤迁移的速度，避免把原服务端的网络打满，使复制时间较长；另外，整个流程涉及的模块比较多，代理层对客户端性能也有一些影响；扩容后有可能在某些异常情况下，客

户端的请求仍然会访问老的分片，但是服务端由于没有slot信息，无法判断该Key是否应该属于自己，便会导致数据混乱。

为了解决以上问题，开始对服务端进行改造，服务端引入slot的概念，数据按照slot进行组织，使一个服务端实例可以服务多个slot，且客户端中维持一份与服务端一致的slot信息（如图3-10所示）。由于服务端有了slot信息，因此可以判断某个Key是在自己的服务范围内还是已经迁移出去了，可以避免数据被写错。有了slot，在迁移时，服务端便可以以slot为单位进行数据迁移，避免了要从全部数据中找出需要迁移的数据，另外可以直接通过待扩容的服务端往新扩容的服务端写数据，避免数据需要通过过滤层的转发，也不需要多次对数据进行序列化和反序列化，大大提高了迁移速度。

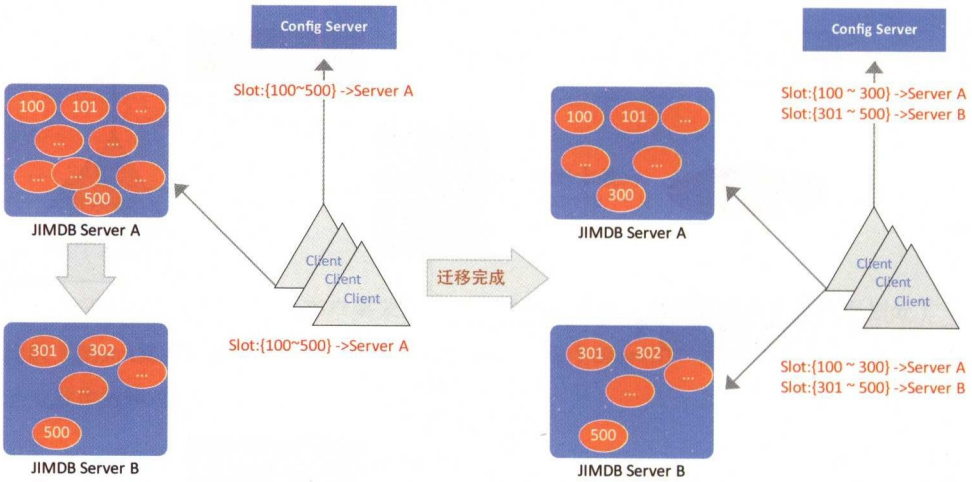


图3-10　基于slot扩容

第二版

随着集群规模进一步扩大，使用的业务越来越多，很多以前注意不到的问题逐渐暴露出来。

自动弹性调度

业务流量突然飙升、容量不足等都需要运维通过管理工具进行扩容增加实例数，另外也有一部分业务申请了集群空间，但由于业务调整等原因，访问量变小了或者停用了，平台管理人员比较难发现这种集群利用率降低的情况。为了提高平台自动化的能力，减少运维人员的工作量，需要让平台动起来，于是弹性伸缩的需求摆在了开发人员的面前。

为了让平台弹性伸缩起来，需要对集群的各项指标进行监控，比如OPS、内存使用率、网络流量等，统计这些指标是否在一段时间内达到了设置的阈值。当超过扩容的阈值时会自动触发扩容；当低于缩容的阈值时会自动进行缩容释放资源（如图3-11所示）。

缩容的过程和扩容的过程基本一致。扩容是把一个实例上的部分slot迁移到新的实例上，缩容是把一个Shard实例上的所有slot迁移到另一个实例上进行合并。

扩容时需要增加实例，那么增加的实例部署在哪台机器上才合适呢？为了选择出最优的机器，有一个采集程序会定期进行信息收集，然后根据CPU繁忙情况、网络流量、OPS、内存剩余空间、机器上的实例数等进行综合打分，各项指标都比较空闲的得分高，如果有一项指标不符合部署要求则直接淘汰，然后再从得分高的机器中选择一台机器进行部署。由于扩容在集群中是并发进行的，因此多个处理线程有可能同时把实例部署到同一台物理机上，当大家部署完成后，实例数等指标可能就不符合要求了，因此需要有一个预分配资源的计算，对未使用的资源进行预占并被计算在内。如果部署失败，则需要把这些资源值做相应的扩除，避免并发部署出现使用资源超限的情况。

对同一个集群还需要控制每台物理机上最大可部署的实例数，避免同一台物理机部署实例数过多，当机器故障时造成对同一个集群影响过大的情况。

为了防止同一间机房路由器故障或者断电等情况的出现，同一个Shard的主从实例应该跨机架；对有跨机房需求的应用，同一个Shard的主从实例还应该部署在不同的机房。

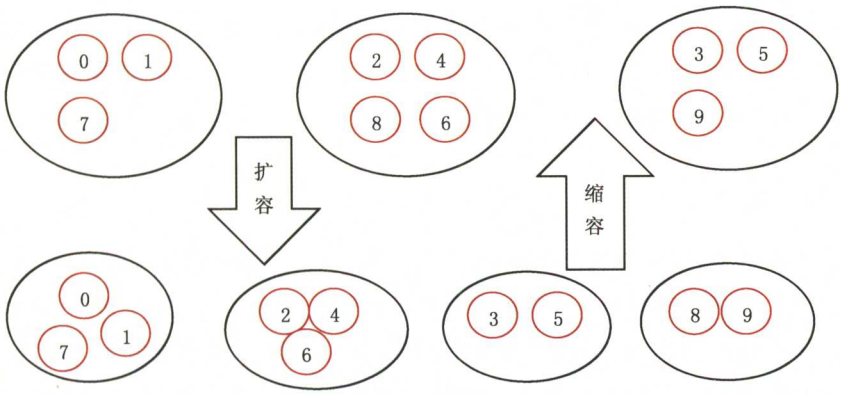


图3-11　扩容与缩容

服务端升级

一个平台需要维护成千上万个进程，这些进程在升级过程中不能中断服务，而且这些进

程还是有状态的，必须保证数据正确无丢失，不能通过简单的流量切换来解决问题，升级程序成为了一项富有挑战的事情。同一台机器由于服务端升级需要同时运行多个版本，程序运行文件的分发和维护也变得不好管理，容易引起混淆，如果由运维人员手动处理，不但时间会变得非常漫长，而且容易出错。

为了解决版本控制和快速创建、销毁服务的问题，引入了Docker，通过Docker的registry进行服务端版本管理和分发。通过调用docker deamon进程的接口启动一个Docker容器，创建一个JIMDB的服务端实例，销毁时也调用docker deamon程序销毁容器。

程序升级过程中涉及数据的复制。为了加快升级速度，考虑到升级完成后旧的实例就可以销毁，因此在同一台物理机上创建一个新版本的实例，当旧实例销毁后并不会导致物理机上实例数超限等问题，通过数据复制流程把旧实例上的slot全部迁移到新的实例上，由于数据在同一个物理机上流动，因此速度会比网络传输快很多。多个集群可以同时进行，也不会导致网络流量由于数据的迁移而暴涨。

资源隔离

不同的业务呈现出不同的业务特点，有的数据量不大但是并发量大，有的数据量比较大但是访问相对不频繁，有的业务写入并发量大读少，有的业务间歇性访问量大，有的持续性访问量大，业务之间的重要程度也不尽相同，对性能的要求也不同。平台在部署集群时需要考虑这些因素，按照业务情况划分不同的资源分区，对性能要求高的、业务重要的集群，其部署的分区需要控制每台物理机部署的实例，单台物理机上的总实例数少一些。对重要程度相对低一些的集群，其部署分区中的每台物理机可以多部署一些实例，实例数可以超过CPU的逻辑核数。管理端需要对物理机划分不同的分区，也需要给集群划分分区，在分区上可以设置单台物理机上最大可部署的实例数，以及同一集群部署在同一物理机上的最大分片数。

大Key扫描

由于Redis单进程及单线程处理请求的特点，不适合一次处理的数据量过大，占用CPU时间过长的请求，比如频繁读取以MB为单位计算的数据量，或者一次从集合中取出成千上万条数据项，任意删除一个特别大的集合都可能导致服务端处理几十毫秒或几秒，从而导致其他客户端的请求被堵塞，TP99性能变差，对性能要求严格的业务会直接导致访问超时。为了协助业务发现这些问题，除了定期地扫描slowlog外，还需要通过外部程序去扫描值比较大或者数据项比较多的集合，然后通过邮件等形式告诉业务负责人，这样业务可以及时地调整，避免在大促时流量增长后才发现问题，那样调整就已经来不及了。

读策略

当业务的读请求OPS非常高时，如果简单地通过分裂扩容，降低单个Shard服务的slot数量，对资源会有一些浪费，在业务允许有一定写延迟的情况下可以通过添加多个从的方式扩展单个Shard读服务的能力。另外，当单个Key读访问量比较大时，扩容并不能解决问题，添加多个从，让客户端的访问压力分摊到多个实例上解决热Key的问题。读取访问可以有多种策略，比如Master优先，当Master访问异常时再访问Slave，也可以轮询多个Slave，或者随机访问一个实例。

3.2.3 案例分享

单次取出集合中的全部数据项

应用在日常运行过程中，单个Key对应的List列表并不长，开发人员采用“lrange key 0 -1”获取全部的数据项，运行平稳。有一天，另外的同事对业务进行了调整，单个Key中的数据项暴涨超过了几十万条，单个查询严重堵塞其他命令。

建议：如果业务能限制单个Key对应的长度最好，如果不行，则需要限制单次取出的数据项数量，分批执行。如果数据项过多、Key过期或主动删除单个Key也会导致命令堵塞。

单个VALUE过大

业务在开发时为了使用上的简单，可能存放一个比较大的VLAUE，比如几MB。访问频率低时没有发现问题，但是当业务新上线一个程序，增加了对这个Key的访问次数时，相应接口的TP99便从2毫秒上升到了几秒。

建议：业务拆分VALUE。

警惕单Key写操作与业务增长呈线性或几何级增长

统计访问流量，常用的方式就是访问一次、调用一次自增。但是当访问量巨大时，会导致单个Key的访问性能根本达不到业务的要求。而且单个Key写操作频繁，无法通过扩容、增加Slave等手段应急解决，只能业务修改代码（大促时只能眼看着服务被“打死”）。

建议：缓存操作是快，但是单个实例的服务能力也是有上限的，需要在业务设计时绕开。比如，拆成对多个Key的写，让流量分摊到多个Shard上去，读的时候再汇总等。

热Key本地没有做缓存

修改不频繁、调用频繁的Key在本地没有缓存，每次都通过远程获取，流量上去后，TP99性能变差。

建议：对性能要求严苛的应用需要考虑在本地应用中进行缓存。

从2014年年初JIMDB立项至今，经过团队的不懈努力和持续优化，JIMDB从最初几台机器发展到今天数千台服务器、多个IDC的部署规模，支撑了包括搜索、推荐、商品详情等京东商城几乎所有的动态内容存储，如数据库领域泰斗Jim Gray所言“内存是新的磁盘”，JIMDB一直见证并践行着这个观点。

3.3 FBase：大表存储

3.3.1 背景

在运营JIMDB的过程中发现，业务系统除了有持久化的需求，还希望可以通过记录的VALUE部分的内容进行条件过滤。JIMDB可以应对非常频繁的业务请求，但是不能进行条件过滤和范围查询；HBase可以满足特定的过滤条件和范围查询，但是不适合作为在线业务使用。为了解决这些问题，我们决定构建一个K-V持久化的存储系统，它可以满足以下5个基本的需求。

◎可以满足在线业务的大并发读写需求。

◎除了支持按Key取值外，还可以支持范围扫描。

◎可以持久化存储，数据不丢失。

◎支持宽表，业务系统可以非常方便地增减column。

◎自动地故障恢复和自动扩容。

3.3.2 系统设计

FBase有schema的概念，其定义类似数据库的Table，每个Table理论上支持无限长的column，每个column有数据类型的定义，与数据库的Table不同的是，schema支持定义column

模板，业务系统在插入记录时，发现column名字不存在，就会根据名字去找模板，如果匹配上了，就会新增一个column，类型与模板定义的类型相同。Table的数据存放在Range中，每个Range只存放一个连续范围的数据，Range之间的数据范围不重叠。当然，如果对范围扫描没有需求的业务，为了写入时分散，可以在主键Key之前拼接一个基于Key的Hash槽信息。目前，FBase的Table只支持一个主键索引，这个主键索引也就是这条记录的Key。但是与HBase不同的是，这个主键索引的Key可以是多个独立的column组成的，column的数据类型可以不同。FBase的表结构如图3-12所示。

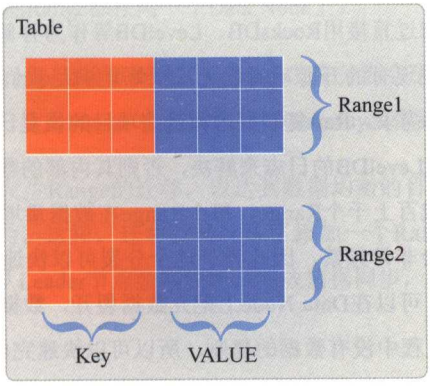


图3-12　FBase表结构

整体设计

FBase的整体架构图如图3-13所示。

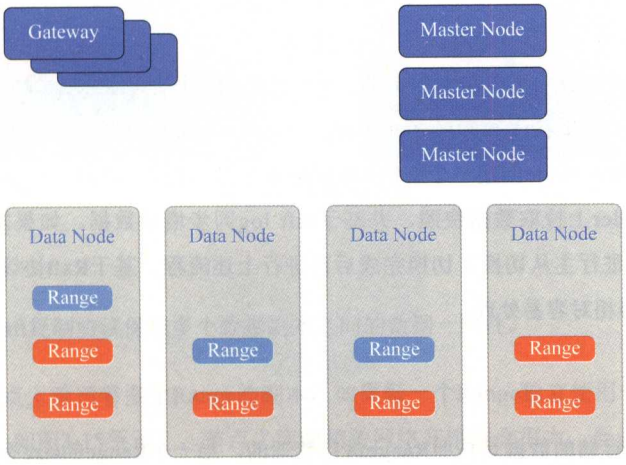


图3-13　FBase架构图

◎Gateway：是业务接入层，当前支持MySQL协议，可以使用MySQL驱动进行接入。

◎Master Node：提供集群元数据管理和存储服务；另外还提供故障检测和扩容决策等，元数据通过Raft协议在Master Node上存储多份。

◎Data Node：业务数据存放的地方，每张业务表被分成了*N*个Range，每个Range可以有多个副本，数据副本之间通过Raft协议保持一致。

存储引擎

在系统设计之初，考虑过直接用RocksDB、LevelDB等作为存储引擎，避免重复造轮子，但是随着设计的深入，发现现有的开源项目已无法很好地满足我们的需求，这些项目设计之初并没有考虑复制、扩容等需求。Raft复制日志和这些项目的恢复日志存在重复，但是又不能通过简单地关闭RocksDB或LevelDB的日志来解决，否则其内部的数据恢复机制会受到影响。一个数据节点上希望存放成百上千个Range，每个Range在数据量增长达到一定的阈值时能够自动分裂，由于分裂活动会非常频繁，因此为了这个分裂可以快速进行，要尽量减少数据的复制。自己开发存储引擎，可以在Data Node上把元数据裂开，数据文件通过文件链接的方式进行引用，在这个分裂的过程中没有数据的复制，所以可以快速完成，不属于新Range的数据可以在compact阶段删掉。分裂后的Raft日志和新写入的数据分别写入新的Range中，数据节点存储空间不够或者进行负载均衡时可以以Range为单位进行迁移。

新的存储引擎借鉴了LevelDB的思想，基于Go语言进行了重写，对Range分裂场景进行了优化，对recover机制进行了调整，并基于raft log进行数据恢复。

Failover

当一个Raft组中有节点发生故障，在一段时间内持续没有上报心跳信息时，Raft组中Leader所在的Data Node会上报给Master Node。Master Node会创建一个补Raft节点的任务，任务会通知新的Data Node创建一个新的Range副本并加入到该Raft组中，新加入的成员会通过Raft协议从Leader上拉取数据快照，并基于raft log同步增量数据。如果发送故障的是raft leader，那么会先进行主从切换，切换完成后再进行上述流程。基于Raft协议，故障的恢复和主从的切换会变得相对容易处理。

扩容

Data Node上存储的数据是按照Range进行管理的，当一个Range的size不断增大，达到分裂阈值时，就会触发分裂，一个Range分裂成两个新的Range，原来的Range被销毁。

Data Node定时向Master上报Range当前的大小，如果Master认为某个Range需要分裂，就会生成相关的调度任务，通知Data Node分裂Range。

Range的分裂采用本地分裂方案，相比其他的开源实现，我们对分裂做了特殊优化，分裂的速度非常快。因为是本地分裂，所以我们在分裂时首先禁止对Range读写，然后将memtable中的数据flush到磁盘，新分裂的Range依赖的数据文件是通过对目标Range的数据文件的链接找到的（如图3-14所示），新分裂Range的store启动后台任务异步执行compact，删掉不属于自己的数据，生成新的数据文件。分裂时，阻塞的请求在分裂完成后可以继续在当前Node上进行处理，新的Range和原来的Range都在同一个Data Node上。

分裂完成后，原来的Range就可以删除了，Range发生分裂会引起Range拓扑的变化，Data Node会主动上报分裂完成，然后Master 会更新拓扑，Gateway会拿到最新的拓扑信息。

分裂完成后，一般会发生Range的迁移，以达到数据均衡的目的。Range迁移的核心是通过Raft成员的变更来完成的，这样，迁移就转化成了添加一个Raft成员，删除一个现有的成员。新加入的成员会异步从Leader节点拉取快照，完成数据同步，同步完成后会通知Raft组把现有的一个成员移除。

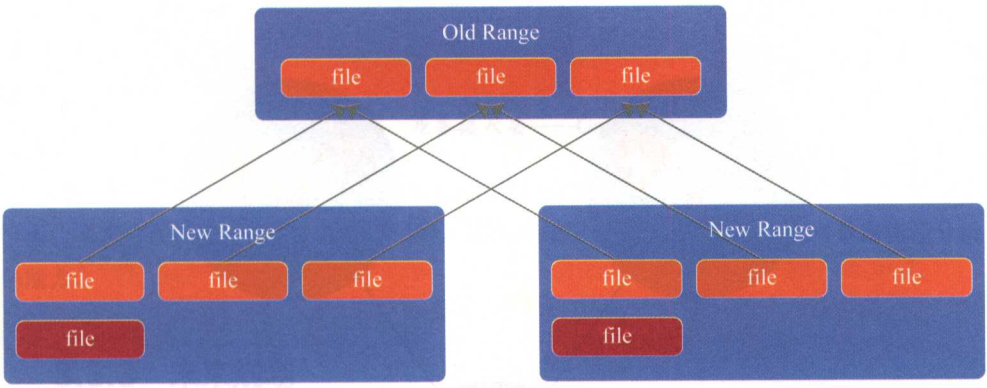


图3-14　Range分裂

Raft

FBase采用Raft复制协议保证多个数据副本之间的数据一致性。

一个存储节点上管理着多个Range的副本，如果每一个Range独立使用一套Raft组件的话（即Single Raft，如图3-15所示），节点之间的相关网络开销就会很大，会占用大量的网络带宽。因此我们采用改良的Raft组件Multi Raft（如图3-16所示），统一代理节点上副本的心跳，

减少Raft心跳带来的网络开销。

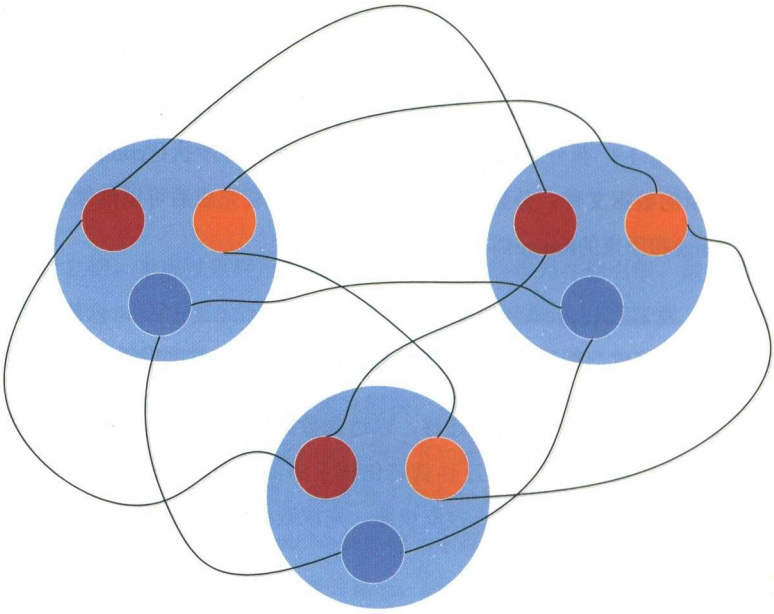


图3-15　Single Raft

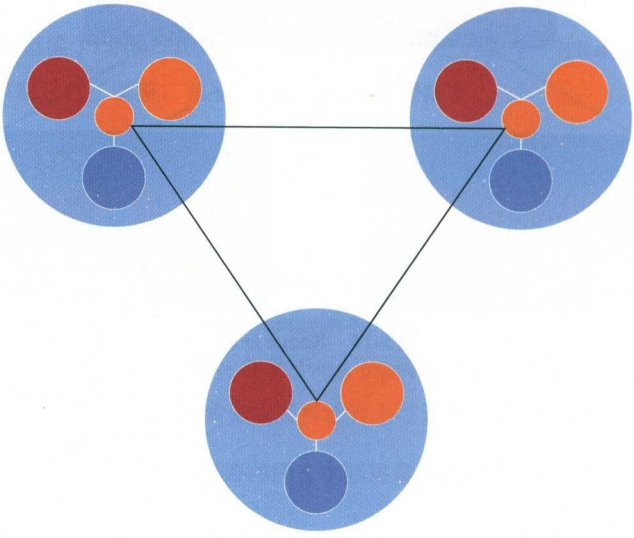


图3-16　Multi Raft

Gateway

网关节点（Gateway）为业务的接入层（如图3-17所示），为了方便推广和减少不同语言客户端的开发成本，目前采用兼容MySQL协议的方式，通过SQL进行相关的操作。当前实现的SQL语句非常简单，只允许单表查询，进行简单的条件过滤。通过INSERT语句完成数据的写入，DELETE语句实现数据的删除，不支持数据更新，如果写入的Key相同则会发生覆盖行为。目前只保证单条的事务，不提供跨行事务。

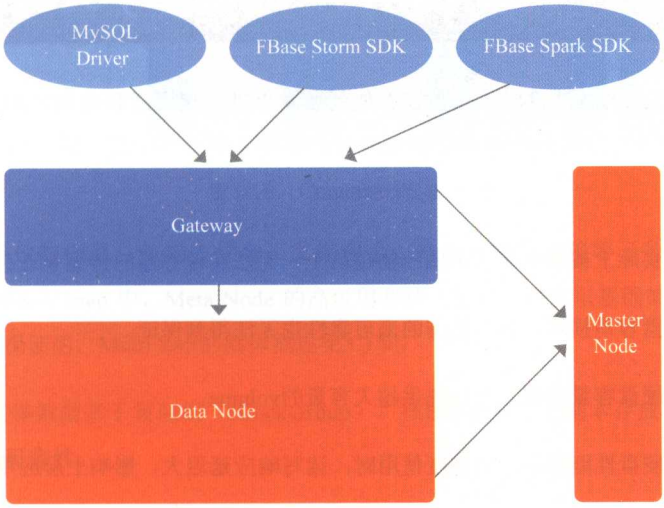


图3-17　业务接入层——Gateway

为了方便大数据处理相关的框架，可以使用FBase，提供与Storm、Spark等平台集成的SDK。未来会提供基于列存储的存储引擎，提高存储层压缩能力，提高系统对数据统计分析的能力。

3.3.3　未来规划

当前支持的业务场景相对比较简单，未来希望在功能和性能上可以提高，简化业务使用成本，提高开发效率。具体规划有如下4项。

◎除了主键索引外，其他column还可以创建其他的索引，方便查询过滤。

◎支持多行数据最终一致性，使业务在插入多行数据时不需要自己处理部分失败的场景。

◎提供列式存储引擎，更高的压缩比，提高统计分析等场景下的访问性能。

◎目前还只支持按范围切分Range。在一些场景下，一层的数据分裂模式可能并不是非常合适，正在开发支持二层的数据分裂模式，通过Hash与范围路由嵌套，提高写入与单Key的读取性能。

3.4 Container File System

随着JDOS系统的持续推广，对数据存储的要求也越来越灵活和苛刻。JDOS 1.0时代是使用宿主机的磁盘资源来提供容器数据存储的。该方案简单易实现，但也有一些明显的弊端，如下。

◎容器数据实际上被捆绑在了所属的物理机上，无法实现容器的伸缩和迁移。

◎物理机硬盘容易损坏，容器数据的高可靠性也无法得到保证。

◎单物理机硬盘容量有限，无法提供超大容量的volume。

◎单物理机硬盘性能有限，当重度使用时，读写响应延迟大，影响上层应用使用。

Container File System （简称ContainerFS）是针对JDOS 2.0 系统开发的一个分布式文件系统，同时适用于原生Kubernetes集群及其他应用场景。ContainerFS，为容器而生。

3.4.1 核心概念

ContainerFS的核心概念是volume = a MetaData Table + Multiple BlockGroups。ContainerFS架构图如图3-18所示。

系统分为四大部分，包括 Meta Node、Data Node、Volume Manger、Client。

每个 volume 有一个自己的namespace，包括1个元数据表和多个BlockGroup，根据集群的规模，可以创建众多大小不一的 volume。每个 volume 都是一个 ContainerFS 的实例，拥有自身文件系统属性，支持标准POSIX协议，可被fuse挂载。

volume 通过客户端挂载之后，即具备了文件系统功能，可以当作本地目录使用。

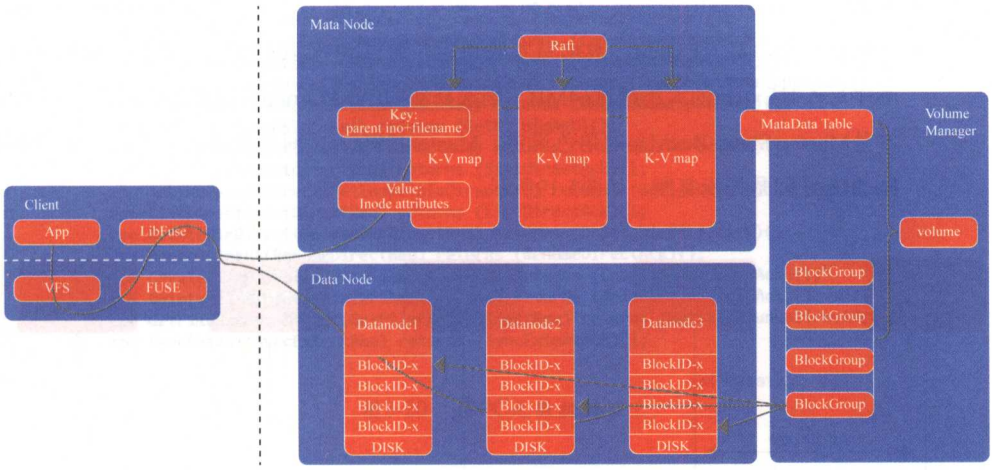


图3-18　ContainerFS架构图

文件的元数据信息，包括文件名、文件大小、修改时间、链接等元数据信息，存储在Meta Node内存K-V map 中，Meta Node 的高可用和持久化是通过京东基础架构部出品的Multi Raft 复制协议保证的，Multi Raft的具体描述见3.3节。

文件的实体数据基于复制组（BlockGroup），按Extent分三个副本存放在Data Node中，保证数据的高可靠性。

通过上述架构，每个volume都可以被一个或多个容器直接挂载使用，volume 的个数根据集群规模可以创建成千上万个，且性能优秀。

3.4.2　核心流程

Container FS的核心流程有如下3个。

1. Open：应用程序通过客户端将某个volume挂载到本地的某个目录，比如 /mnt，客户端通过fuse接管了用户的打开文件、读写文件等操作。当用户打开一个文件时，客户端通过元数据gRPC接口获得inode，并将InodeInfo写入Meta Node中，Meta Node会向客户端返回打开文件成功的消息，客户端会生成一个文件句柄返回给fuse客户端，客户端将该Dirent信息更新到缓存，这个流程完成了打开文件的流程。

2. Write：应用程序调用Write接口写入数据，按实际写入量，在归属该volume的BlockGroups中选择一个，写入Extent，数据写入成功后，通过gRPC接口更新元数据。如果文件顺序写入时Extent的大小超过64MB，则选择新的BlockGroup写入新的Extent。三个副本是Master-Slave-BackUp的角色设定，Master和Slave之间是流式同步复制，Slave和BackUp之间是异步复制，如图3-19所示。

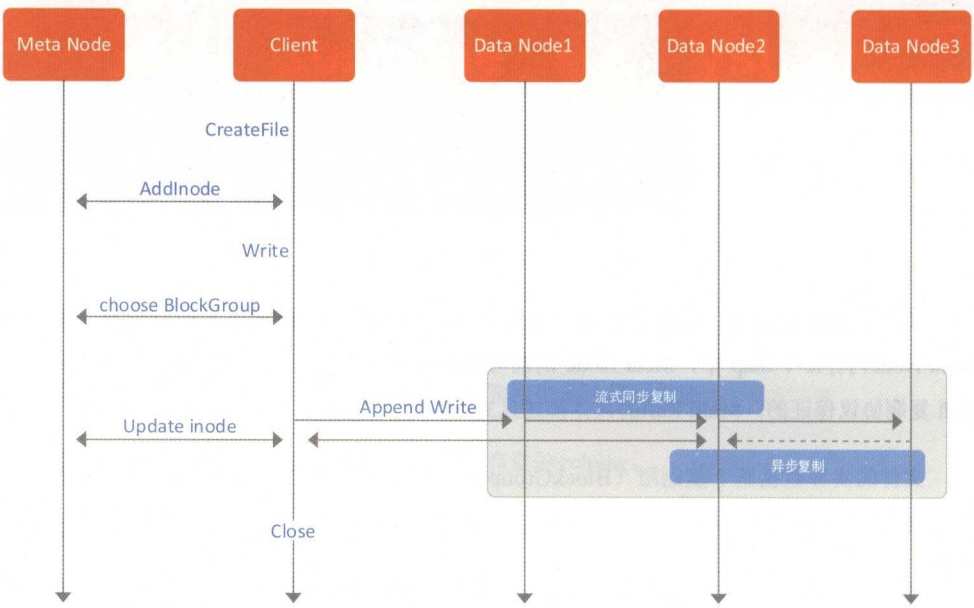


图3-19 写文件流程

3. Read：用户可以在其他的物理机或者容器中并行多次挂载该volume，应用程序从挂载的目录中读取文件，首先会去Meta Node获取文件元数据信息，包括文件大小、文件Extent等信息，获取到这些信息后，客户端就可以定位到该文件的实体数据分布在哪些Data Node上，并根据用户读文件的偏移量，来定位该从哪个BlockGroup 的哪个Extent开始读取。volume是可共享的，多个挂载点可以同时读取相同或不同的文件。

目前，ContainerFS系统的客户端通过Go语言实现，在Meta Node中实现了POSIX协议的全部接口，并基于gRPC将元数据接口标准化，支持各类语言的客户端定制开发。gRPC接口如图3-20所示。

```
service MetaNode {
    rpc GetMetaLeader(GetMetaLeaderReq) returns (GetMetaLeaderAck){};
    rpc CreateNameSpace(CreateNameSpaceReq) returns (CreateNameSpaceAck){};
    rpc ExpandNameSpace(ExpandNameSpaceReq) returns (ExpandNameSpaceAck){};
    rpc SnapShootNameSpace(SnapShootNameSpaceReq) returns (SnapShootNameSpaceAck){};
    rpc DeleteNameSpace(DeleteNameSpaceReq) returns (DeleteNameSpaceAck){};
    rpc GetFSInfo(GetFSInfoReq) returns (GetFSInfoAck){};
    rpc CreateDirDirect(CreateDirDirectReq) returns (CreateDirDirectAck){};
    rpc StatDirect(StatDirectReq) returns (StatDirectAck){};
    rpc GetInodeInfoDirect(GetInodeInfoDirectReq) returns (GetInodeInfoDirectAck){};
    rpc ListDirect(ListDirectReq) returns (ListDirectAck){};
    rpc DeleteDirDirect(DeleteDirDirectReq) returns (DeleteDirDirectAck){};
    rpc RenameDirect(RenameDirectReq) returns (RenameDirectAck){};
    rpc CreateFileDirect(CreateFileDirectReq) returns (CreateFileDirectAck){};
    rpc DeleteFileDirect(DeleteFileDirectReq) returns (DeleteFileDirectAck){};
    rpc GetFileExtentsDirect(GetFileChunksDirectReq) returns (GetFileChunksDirectAck){};
    rpc SyncExtent(SyncExtentReq) returns (SyncExtentAck){};
}
```

图3-20　gRPC元数据接口

3.4.3　核心数据结构

核心数据结构如图3-21所示。

```
message Dirent{
    bool InodeType = 1;
    uint64 Inode = 2;
}
message InodeInfo{
    int64 CreateTime = 1
    int64 ModifiTime = 2;
    int64 AccessTime = 3;
    uint32 Link = 4;
    int64 FileSize = 5;
    repeated Extent Extents = 6;
}
message Extent{
    uint64 StartOffset = 1;
    uint64 EndOffset = 2;
    uint32 BlockGroupID = 3;
}
message BlockGroup{
    uint32 BlockGroupID = 1;
    int64 FreeSize = 2;
    int32 Status = 3;
    repeated BlockInfo BlockInfos = 4;
}
message VolInfo {
    string VolID = 1;
    string VolName = 2;
    string MetaDomain = 3;
    int32  SpaceQuota = 4 ;
    int32  InodeQuota = 5 ;
    repeated BlockGroup BlockGroups = 6;
}
```

图3-21　核心数据结构

数据结构中相应部分的说明如下。

◎Dirent 体现Inode和InodeType (文件或者目录)。

◎InodeInfo体现具体文件或者目录的属性，包括文件大小、创建时间等。

◎Extent体现某个文件块，记录了归属的BlockGroupID和文件块在文件中的偏移量。

◎BlockGroup和VolInfo 体现了某个volume的具体信息。

元数据的索引关系如下。

◎Parent Inode + Name → Dirent。

◎Inode → InodeInfo。

◎Extent → BlockGroup。

客户端通过ReadDirAll 、Lookup 根据应用文件的路径和名称定位到InodeInfo。

3.4.4 产品特性

ContainerFS的产品特性如下。

◎无缝集成：支持标准的文件访问协议，支持fuse挂载，业务应用无须任何修改即可无缝使用。

◎共享访问：共享访问帮助多个业务应用获得相同的数据来源。

◎弹性伸缩：可满足业务增长对文件存储的容量诉求。

◎线性扩展的性能：线性扩展的存储性能，非常适合数据吞吐型的应用。

目前，业内也有其他分布式文件系统，可勉强为小规模集群的容器提供数据存储空间，但都有自己不可避免的缺点。最知名的有两个，分别是 Red Hat公司的GlusterFS方案和 CoreOS公司的Torus方案。两种方案具体说明如下。

◎GlusterFS方案：该方案使用GlusterFS分布式文件系统，创建一个逻辑硬盘，挂载到容器所在的物理机上，并映射给容器使用。GlusterFS的设计决定了volume的数量无法分配很多，它更适用于需要一块大容量的存储空间，而不适用于容器集群，当容器集群扩大，需要成千上万个volume时，GlusterFS根本无法提供这种服务。

◎Torus方案：该方案使用Torus分布式块存储，利用NDB在容器所在的物理机上创建一个块设备，并通过格式化和分区映射给容器使用。该方案虽然解决了GlusterFS方案的弊端，但它是利用NDB提供块设备的方式来提供volume的，自身没有文件系统属性，这决定了该方案提供的存储空间无法实现数据共享和并发读写等高级特性，而且Torus项目目前处于较初期的阶段，还无法用于生产环境。

3.4.5　典型应用

ContainerFS典型应用1，如图3-22所示。

作为JDOS 2.0 的数据存储引擎，ContainerFS提供了独享、共享等类型的volume，并通过PV机制挂载给POD或者容器使用。其使用效果如图3-23所示。

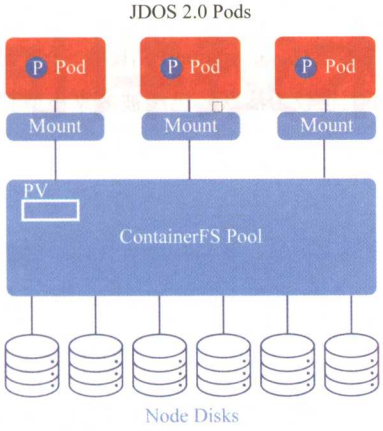


图3-22　典型应用1

```
[root@node-218 mnt]# df -h
Filesystem                                 Size  Used Avail Use% Mounted on
/dev/mapper/centos-root                    50G   11G   40G  21% /
devtmpfs                                   16G     0   16G   0% /dev
tmpfs                                      16G     0   16G   0% /dev/shm
tmpfs                                      16G  570M   16G   4% /run
tmpfs                                      16G     0   16G   0% /sys/fs/cgroup
/dev/mapper/centos-home                   771G   16G  755G   3% /home
/dev/sda1                                 497M  160M  331M  34% /boot
tmpfs                                     3.2G     0  3.2G   0% /run/user/0
ContainerFS-6636e72740f1371080ebc4e0c4704a79  20G  17G  3.9G  81% /tmp/mnt
ContainerFS-38d85dab40be8333806ce39f3101173ae  20G     0   20G   0% /home/kubelet/pods/21facc89-41e7-11e7-876d-e0db55137a74/volumes/kubernetes.io~containerfs/pvc-7d5448cb-41e2-11e7-876d-e0db55137a74
ContainerFS-38d85dab40be8333806ce39f3101173ae  20G     0   20G   0% /home/kubelet/pods/9afa03a2-41e8-11e7-876d-e0db55137a74/volumes/kubernetes.io~containerfs/pvc-7d5448cb-41e2-11e7-876d-e0db55137a74
```

图3-23　使用效果

ContainerFS典型应用2，如图3-24所示。

作为大数据存储引擎，ContainerFS为数据分析提供存储资源池、存放原始数据、训练数据及数据报告。

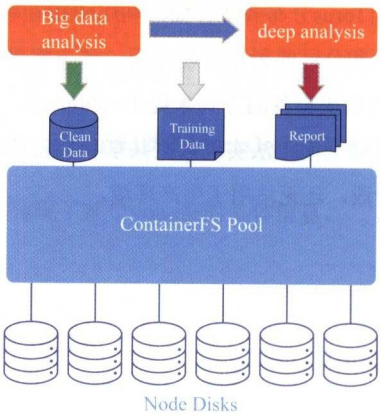
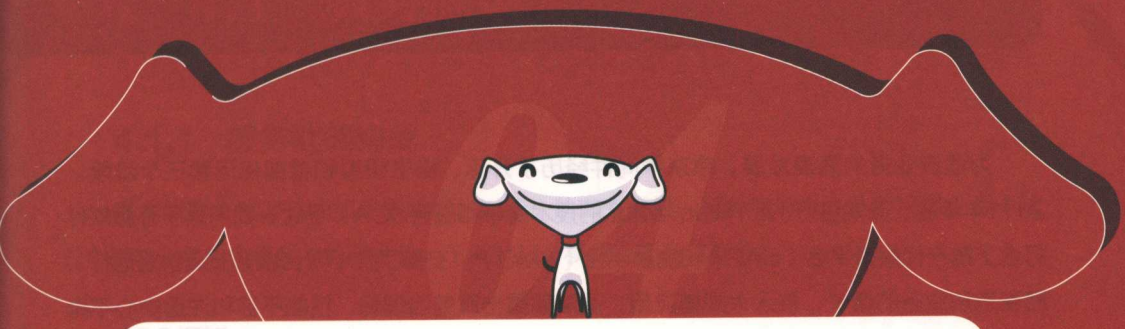


图3-24 典型应用2

3.4.6 小结

以上就是ContainerFS的基本信息，ContainerFS作为一个开源项目，在承载京东内部业务的同时，目前在社区和GitHub上也较为活跃，已经被多家小型互联网公司使用，社区也在不断壮大中。



第 4 章
中间件技术

4.1 服务框架

4.2 消息队列

4.3 JMQ 复制技术解析

4.4 CallGraph：分布式服务跟踪系统

为适应业务的高速发展，京东中间件经历了开源、自主研发和精细化运营三个阶段。2011年年底，开始进行.NET转Java体系的技术变革和服务SOA。当时，我们基于开源软件打造了第一代服务平台（SAF）和消息平台（AMQ），支撑了几十万台服务节点稳定运行。随着京东业务的增长，接入大规模应用，逐渐暴露出服务治理弱、性能不够和注册中心不稳定等诸多痛点。因此，从2014开始，我们开始自主研发了功能更加强大的中间件平台，包括微服务平台（JSF）和消息平台（JMQ）。新平台历经数次618和双11大促的考验，不断根据业务需求和线上问题进行迭代完善，目前已经非常成熟稳定，覆盖了公司全业务链。

JSF服务节点总数有200多万，日常调用总量达到了2000亿次。如此庞大复杂的服务体系，很难清楚准确地描述其整体架构脉络和调用链路，定位问题更是费时费力。随后，我们又自主研发了分布式服务跟踪系统，提供了黄金链路视图、清晰的调用链关系视图及精准便捷的故障定位。

JMQ具有集群高可用、数据高可靠、支持集群化管理等特性。不仅考虑了高吞吐、高可用、低运维成本等方面的需求，同时还针对京东业务场景增加了重试、延时消费、归档及事务等业务方面的需求。使用Zero-Copy、日志和队列文件、组提交、内存映像文件及优化的复制协议等技术使TPS的性能比AMQ提升了近10倍。

现阶段，为配合京东异地多活整体战略，我们正在进行服务平台系统的升级改造，提升系统的灾备能力和用户体验。

回顾每次历程，不仅是对架构设计和技术实现的挑战，也是对研发人员意志力和专注力的考验，更是为我们的下一次技术演进奠定了基础，正所谓既是终点又是起点，我们将在实践中不断探索适合京东业务发展的技术架构之路。本章将主要介绍京东中间件在不同阶段的相关背景，解密其背后的技术选型和设计思路。

4.1 服务框架

4.1.1 服务框架构成

基本的服务框架包括如下三个模块：统一的RPC框架、服务注册中心及管理端。有了这三个模块，就能实现基本的服务化。下面对三个模块进行具体分析。

RPC框架

RPC，即远程过程调用，Remote Procedure Call的缩写。选择RPC框架的考量标准如下。

◎代码规范：例如是对已有代码透明，还是代码生成。

◎通信协议：例如是TCP还是HTTP。

◎序列化协议：例如是二进制还是文本，是否需要跨语言，性能如何。

◎I/O模型：异步/同步，阻塞/非阻塞。

◎负载均衡：客户端软负载，代理模式，服务端负载。

另外，如果是从开源里面选择RPC框架，那么还需要考量以下4点。

◎成熟度：包括学习成本、社区热度、文档数、是否有团队维护，以及稳定性（盲目追求的不一定是最适合的）。

◎可扩展性：是否有SPI支持扩展，是否支持上下兼容。

◎跨语言：是否支持跨语言。

◎性能：作为基础软件，性能必须出色。

表4-1 RPC开源框架对比表

	Thrift	RESTful	Dubbo	gRPC
代码规范	基于Thrift的IDL生成代码	基于JAX-RS规范	无代码入侵	基于.Proto生成代码
通信协议	TCP	HTTP	TCP	HTTP2

续表

	Thrift	RESTful	Dubbo	gRPC
序列化协议	Thrift	JSON	多协议支持，默认Hessian	protobuf
I/O框架	Thrift自带	Servlet容器	Netty 3.0	Netty 4.0
负载均衡	无	无	客户端软负载	无
跨语言	多种语言	多种语言	Java	多种语言
可扩展性	一般	好	好	差

注：此处未列举SOAP、RMI、Hessian、ICE。

关于RPC框架的选型总结如下。

◎如果需要与前端交互，就适合短连接、跨语言的RPC框架，例如RESTful、gRPC等。

◎如果纯粹后台交互，就适合长连接、序列化为二进制的RPC框架，例如Thrift、Dubbo等，会更高效。

◎如果是小公司、新公司从头开始推广服务化框架，就选择规范化的RPC框架，例如Thrift、RESTful、gRPC。

◎如果是已有大量业务代码的再推广服务框架，就最好选择无代码入侵的RPC框架，例如Dubbo、RESTful。

服务注册中心

注册中心相当于服务提供者和服务调用者之间的引路人，其提供服务注册、服务发现、服务状态检测的基础服务，在服务治理中的作用极为重要。

因此，选择注册中心基本从以下3点考量。

◎服务注册：接收注册信息的方式。

◎服务订阅：返回订阅信息的方式，推还是拉。

◎状态检测：检测服务端存活状态。

重点提一下状态检测，因为如果检测不准确而误判，会导致严重后果。例如，ZooKeeper根据服务端注册的临时节点进行状态检测，如果服务端和ZooKeeper之间的网络闪断，会导致ZooKeeper认为服务端已经宕机，从而摘掉这个节点。但是，其实客户端和服务端之间的网络是好

的，这样就有可能把节点全部摘掉，导致无可用节点。

如果是从开源里面选择，那么还需要考量以下几点。

◎成熟度：包括学习成本、社区热度及文档数（盲目追求的不一定是最适合的）。

◎维护成本：注册中心维护。

◎数据解构：是否能快速定位结果，是否能遍历。

◎性能和稳定性。

◎CAP原则[1]：CP（关注一致性）还是AP（关注可用性）。

表4-2 配置中心开源框架列表

	ZooKeeper	etcd	Consul	Eureka
一致性	强一致性Paxos	强一致性Raft	强一致性Raft	弱一致性
数据结构	Tree	K/V	K/V	K/V
通信协议	TCP	HTTP、gRPC	HTTP、DNS	HTTP
客户端	ZooKeeper Client	/	/	Eureka-client
CAP原则	CP	CP	CP	AP

注：此处未列举Redis和MySQL。

服务注册中心的选型总结如下。

◎规模小则选择CP，RPC框架可以直接接入数据源。

◎规模大则选择AP，RPC框架不可以直接接入数据源。

◎存在跨机房、跨地域的尽量不要选有强一致性协议的注册中心。

◎RPC框架必须要有注册中心不可用的容灾策略。

◎服务状态检测十分重要。

管理端

管理端相对简单，要求有可视化服务提供者和调用者即可。

---

[1] CAP原则：指的是在一个分布式系统中，Consistency（一致性）、Availability（可用性）、Partition tolerance（分区容错性）。

4.1.2 完善的服务框架

如果需要一个完善的服务框架，那么必须增加外部模块，常见的模块如图4-1所示。

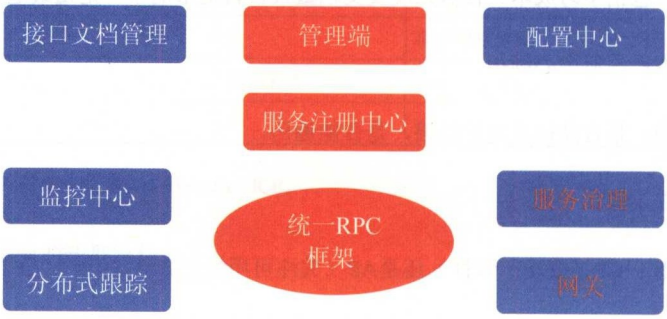


图4-1 完善的服务架构图

接口文档管理

提供一个接口文档管理及接口查询的入口，可以是一个公共的Wiki，也可以是独立的系统，等等。此处定义接口的文档，包括接口描述、方法定义及字段定义。可以定义接口的SLA，包括支持的并发数、TP99多少及建议配置是什么，还有接口的负责人等一些查询的入口。

配置中心

提供一个配置管理的地方，这里说的配置主要指服务相关的一些配置。配置包括分组配置、路由策略、黑白名单、降级开关、限流信息、超时时间、重试次数等任何可以动态变更的数据。这样，服务提供者和服务调用者可以不需要重启自己的应用，直接进行配置的变更。配置中心可以独立于注册中心，也可以和注册中心合并。

监控中心

监控服务关注接口维度和实例（例如所在的JVM实例）维度的数据。

RPC框架可以定时上报调用次数、耗时、异常等信息。

监控中心可以统计出服务质量信息，也可以进行监控报警。

分布式跟踪

区别于监控中心，分布式跟踪以调用链的模式对服务进行性能、调用关系跟踪和分析。

RPC框架作为分布式跟踪系统的一个天然埋点，可以很好地进行数据输出。

服务治理

常见的服务治理功能如下。

◎服务路由。

○权重：例如机器配置高的权重高，机器配置低的权重低。

○IP路由：例如某几台机器只能调相应的几台机器。

○分组路由：例如自动根据配置调某个分组。

○参数路由：例如根据方法名进行读写分类，或者根据参数走不同的节点。

○机房路由：例如只走同机房，或者同机房优先。

◎调用授权。

○应用授权：只有授权后的应用才能调用这组服务。

○token：只有token正确的才调用这组服务。

○黑白名单：只有名单允许的才能调用这组服务。

◎动态分组。

○服务端切分组：可以根据分组的情况，对服务提供者进行一个动态的分组调度。

○客户端切分组：可以对调用者进行一个分组调度。

◎调用限流。

○服务端限流：服务端基于令牌桶或者漏桶模型进行限流。

○客户端限流：根据客户端的标识，进行调用次数限流。

◎灰度部署[1]。

○灰度上线：先启动，验证后再提供服务。

---

[1] 灰度部署：为控制新功能上线影响范围，保证整体系统的稳定，提供了一种平滑过渡的发布方式。

○预发标识：表示该服务为预发布服务。

○接口测试：方便地提供接口自动化测试功能。

◎配置下发。

○服务配置。

○全局配置。

◎服务降级[1]。

○Mock：出现异常或者测试的情况下，返回Mock数据。

○熔断：客户端超时或者服务端超时。

○拒绝服务：服务端压力大时，自动拒绝服务，保护自己。

网关

RPC框架大部分场景都是自己调用的，什么时候会需要一个网关呢？

网关可以提供如下功能。

◎统一的鉴权服务。

◎限流服务。

◎协议转换：外部协议转统一内部协议。

◎Mock：服务测试、降级等。

◎其他一些统一处理逻辑（例如请求解析、响应包装）。

服务注册中心Plus

需要逻辑处理能力，例如对数据进行筛选过滤整合，计算服务路由等功能，同时还需要有与RPC框架交互的功能。

---

[1] 服务降级：当服务器压力剧增时，根据当前业务情况及流量对一些服务和页面有策略地降级，以此释放服务器资源来保证核心任务的正常运行。

管理端Plus

管理端除了之前的简单服务管理功能，还需要提供配置信息展示、监控信息展示及各种维度的数据展示。下面提到的服务治理功能，都可以在管理端进行管理。

另外，常见的服务治理功能，可以提供API给开发人员在程序中调用。

4.1.3 京东实践

第一代SAF背景

2012年年初，京东从.NET转Java。各个部门、各个业务线都没有一个统一的服务化框架，有的是Dubbo、WebService、Hessian等。

同时，各个业务系统自己有非常多的业务代码。通过统计，接口规模在1千左右，服务节点规模在5万左右，机器规模在8千左右，机房比较少，拓扑简单。

因此，当时目标很明确，就如下4条。

◎京东系统服务化、API化的从无到有。

◎统一京东的RPC调用框架。

◎稳定可靠。

◎提供简单的服务治理功能。

第一代SAF选择

结合实际情况和上文总结的一些选型原则，当时的技术选型如下。

◎RPC框架：基于Dubbo 2.3.2做配置扩展，功能扩展包括rest（resteasy）、WebService（cxf）、Kryo/Thrift序列化、调用压缩等。

◎注册中心：选择使用ZooKeeper，RPC框架直接接入数据源。

◎监控中心：监控服务 + HBase。

◎管理平台：读取ZooKeeper做管理平台，提供基本的上下线、黑白名单等功能。

业务系统于2012年4月上线，最大规模时，接口数3千，接入最大IP数2万。

第二代JSF背景

随着京东业务的不断快速增长，接口、机器数也呈数量级增长。多地域、多IDC，部署结构也变得比较复杂。而SAF的一些固有问题也逐渐凸显，如下。

◎RPC框架较重，性能有提高的空间。

◎注册中心无业务逻辑，直接对外暴露。

◎京东复杂的部署架构需要更强大灵活的服务治理功能。

◎监控数据不完整，维度不够。

◎无应用依赖关系。

◎跨语言调用需求。

第二代JSF选择

所以，在2014年年初，我们进行了第二代JSF立项，全部自主研发。

主要的技术决定如下。

◎RPC框架：轻量级、更佳的性能、兼容旧版本协议。

◎注册中心：基于DB作为数据源，前置Index服务，支持十倍接入量，部分逻辑放在注册中心以减少客户端负担。

◎监控中心：监控Proxy服务 + InfluxDB（2015年后改为Elasticsearch）。

◎管理端：基于DB，功能更强大，提供完善的服务治理管理功能；打通京东应用管理平台，提供应用依赖关系梳理。

◎HTTP网关：基于Netty，支持跨语言调用。

开发周期为84人/月（2014年1月到2015年1月），包括开发、测试、预发、上线、推广人员。

JSF架构简图

JSF架构设计充分考虑了京东技术体系的特点，如图4-2所示。

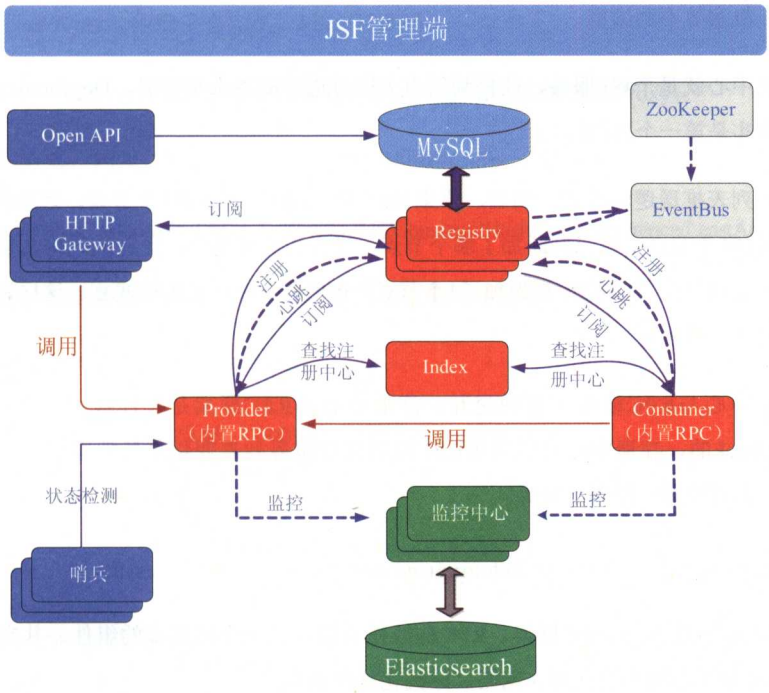


图4-2　JSF架构简图

JSF注册中心

自行开发的Java服务，采用MySQL存储数据。

注册中心主要实现的就是服务列表的注册订阅推送、服务配置的获取下发、服务状态的实时查看等功能。注册中心节点是无状态、可水平扩展的。整个注册中心集群下的所有注册中心节点都是等价的。每个机房部署多个注册中心节点。同机房的RPC框架会优先连接本机房的注册中心节点。

主要亮点如下。

◎引入Index服务概念。该服务就是一个最简单的HTTP服务，用于查找注册中心节点（同机房、压力最小或者其他特定场景），可以认为是不会宕机的服务，RPC框架会优先连接该服务注册中心地址，这样的好处是注册中心地址变化后，RPC框架不用修改任何设置。

◎注册中心内存有服务列表全量缓存，连不上数据库也保证可读。

◎数据库的数据结构更适合各种维度的展示、过滤、分析等，例如根据分组、IP、应

用、机房等不同维度。

◎注册中心就是个JSF服务，监控到压力大即可进行动态水平扩展。Dogfooding、注册中心其实是第一个JSF接口。

◎服务列表推送逻辑改进。例如，原来100个Provider，现在加1个节点，之前的SAF需要下发101个节点，自己判断加了哪个节点，进行长连接建立；现在的改进是修改为下发一个add事件，告知RPC框架加了1个节点，RPC框架进行长连接建立。这样做大大减少了推送的数据量。

◎注册中心与RPC框架可各种交互。注册中心和RPC框架是长连接，而且JSF是支持Callback的，注册中心可以调用RPC框架进行服务列表变化之外的操作。例如查看状态、查看配置、配置下发等。

JSF RPC框架

图4-3所示的是JSF的RPC框架。RPC框架作为服务化里面最基本的组件，其实都大同小异，因为RPC调用都绕不开代理、网络、序列化这些操作。

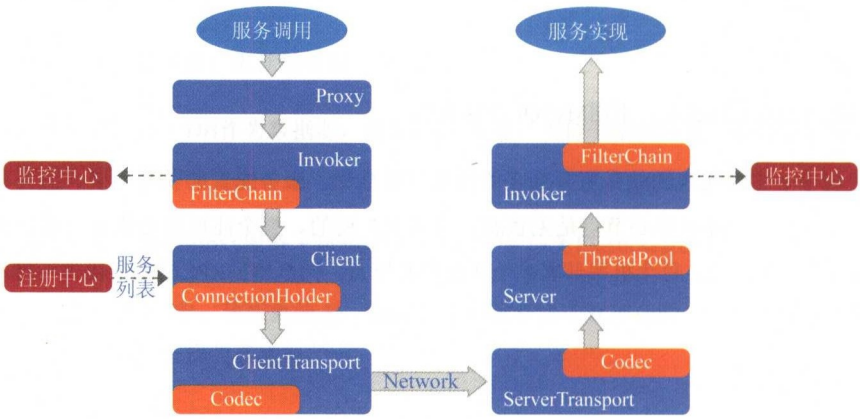


图4-3　JSF RPC框架

JSF的RPC框架也类似，主要分为图4-3中的几个模块。

其功能特性如下。

◎Config：Spring/API/Annotation。

◎Proxy：Javassist/JDK。

◎Invoker/Filter：内置 + 自定义，Filter可扩展。

◎Client：Failover（默认）/FailFast/TransportPinpoint/MultiClientProxy。

◎调用方式：同步（默认）/异步并行/异步回调/Callback/泛化。

◎Loadbalance：Random（默认）/Roundrobin/ConsistentHash/LocalPreference/LeastActiveCall。

◎路由：参数路由、分组路由（IP级别路由逻辑在注册中心做）。

◎长连接维护：可用/死亡/亚健康。

◎协议：JSF（默认）/SAF（Dubbo）/HTTP/Telnet/HTTP2。

◎第三方：REST/WebService。

◎序列化：MsgPack（默认）/Hessian/Json/Java/protobuf（C++）。

◎压缩：Snappy/LZMA。

◎网络：基于Netty 4.0，长连接复用。

◎线程模型：BOSS + WORKER + BIZ。

◎容灾：本地文件。

◎请求上下文：IP、参数、隐式传参。

◎事件监听：响应事件、连接事件、状态事件。

◎分布式跟踪支持：进行数据埋点。

JSF管理平台

提供强大的管理功能，包括服务管理、监控管理、注册中心管理等功能。图4-4展示了服务管理的概况，比如提供者的数目、调用者的数据等；图4-5展示了服务管理的详情，针对每个服务，可以提供“黑白名单”“路由”“上下线”等服务治理功能；图4-6展示了针对服务方法的性能监控，比如调用量等。

图4-4 服务管理概况

图4-5 服务管理详情

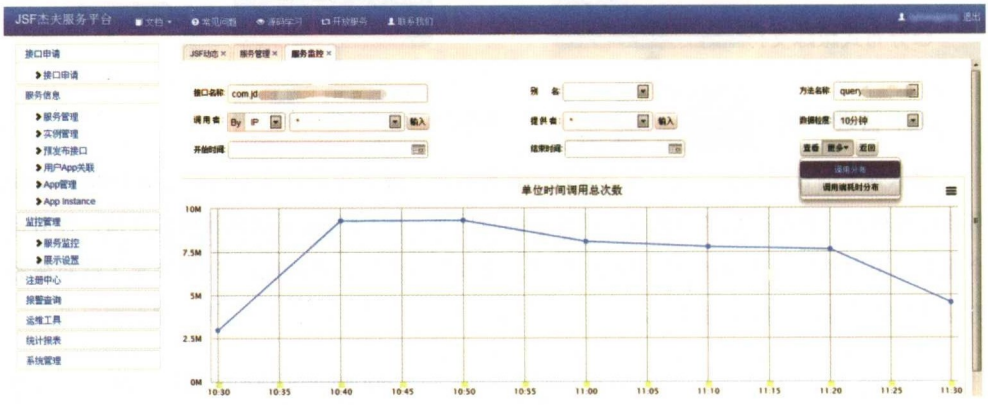


图4-6 服务方法性能监控

针对服务治理的功能，JSF提供了很多API，可以授权给开发人员或者外部系统使用。

例如单元测试调用、限流配置/开关、动态分组、上下线等都提供了开放API。

JSF HTTP网关

网关是为了方便通过HTTP + JSON跨语言调用JSF服务，而不需要使用JSF的RPC框架，其特性如下。

◎基于Netty 4.0实现HTTP网关，没有使用Servlet容器，轻量高效。

◎支持服务自动发现。一般的HTTP服务，为了解决单点问题，都会用“域名 + VIP”等实现高可用、故障转移等；现在，网关同时原生接入了JSF的注册中心，知道了服务的提供者信息（JSF协议支持HTTP调用）。服务提供者也不用关心扩容缩容导致服务的IP端口发生变化，网关会自动维护服务列表。

◎服务限流。针对“方法级 + 应用”进行授权，固定时间只能调用指定次数。同一个方法也只能占用网关内的部分线程。

◎结果统一包装。对异常等响应进行包装。

JSF规模

JSF的各项规模级别如下。

◎接口数：数万级。

◎服务节点数：百万级。

◎接入实例数：数十万级。

◎框架调用量：每天千亿级别。

◎HTTP网关：每天百亿级别。

4.1.4 小结

◎中间件技术领域，保持适度前瞻。设计研发一个系统，必须能够支撑公司未来几年的技术发展。

◎没有最好，只有最适合！不要人云亦云，盲目看大公司用什么，现在什么最新，或者什么性能最好。无论什么时候，都要结合自己的现状及未来几年的规划，来进行技术选型。

◎千里之行，始于足下。服务化框架的工程开发后，是大规模的推广使用，并推动公司业务与开发模式的变革——微服务、组件化、模块化。

4.2 消息队列

本节介绍京东消息队列（MQ）中间件的演进历程，以及在每一代MQ产品中，我们是如何解决MQ面临的一些通用问题的，比如，如何处理I/O、如何存储消息、如何路由消息等。

下面来明确一些基本概念。

◎Broker：消息中间件服务端的一个实例。

◎Producer：消息的发送方。

◎Consumer：消息的接收方。

◎Topic：消息中间件里的数据分类的标示，一个Topic也就代表一类消息，Producer和Consumer通过Topic建立关联。

4.2.1 第一代AMQ

技术选型

◎消息核心：基于ActiveMQ 5.6做扩展。

◎配置中心：MySQL + ZooKeeper。

◎管理监控平台：自主研发。提供比ActiveMQ自带的管理平台更丰富的消息监控功能，并提供消息管理服务。

当时选择ActiveMQ是由于它是由Java开发的，且符合公司的技术路线（.NET转Java），有主从复制方案，性能尚可（单实例TPS 3000+），支持JMS、AMQP等多种技术规范。

AMQ如何解决MQ面临的一些通用问题

因为核心是使用的ActiveMQ，所以部分问题就回归到了ActiveMQ是如何解决这些通用问题上来了，但其中不乏我们对ActiveMQ的扩展，关于ActiveMQ的部分详细情况我们可以参看其官方文档，以下主要是我们在其基础上的扩展及改进。

1．如何存储？

ActiveMQ使用了KahaDB存储引擎进行存储，使用BTree索引来保证可靠性索引文件和日志文件都要同步刷盘。此外，ActiveMQ通过虚拟主题（Virtual Topic）的方式实现Topic，也就是如果一个Topic有多个订阅者，ActiveMQ就为每个订阅者创建一个队列，那么Producer每发一条消息，有多少个订阅者，Broker就会将消息复制多少份，将其放到不同订阅者的队列中，Broker再从中获取消息推送给Consumer。

2．如何支持集群？

当时原生的ActiveMQ客户端在收发消息上是这样的：针对一个Topic，发送者只能往特定的单个Broker上发送消息，消费者亦是如此，只能从指定的单个Broker上消费消息。从客户端的角度来看就是不支持服务集群化，一旦分给这个Topic的这组Broker挂掉，那么针对这个Topic来说整个服务就处于不可用状态。

我们做的第一件事就是让客户端支持集群，我们采用ZooKeeper作为配置中心对客户端进行了扩展，使客户端支持了集群的同时还实现了其对服务端动态扩展的支持。客户端和服务端的整体架构如图4-7所示。

3．push还是pull？消息如何路由？

AMQ采用的是push模式，也就是消息由Producer发送到Broker端之后由Broker推送给Consumer。

对于原生的客户端只能在单个Broker上收发消息，那也就不存在消息路由的问题了。我们来看一下集群化之后消息是如何路由的，这里的路由涉及两个方面：一方面是Producer在发消息时如何选择将消息发往哪个Broker；另一方面就是Broker如何决定一条消息应该推送给哪个Consumer。

发送路由：发送路由有如下两种发送方式。

（1）随机发送。这种模式是完全随机的，也就是Producer在Topic指定的多个Broker当中随机选择一个Broker向其发消息。

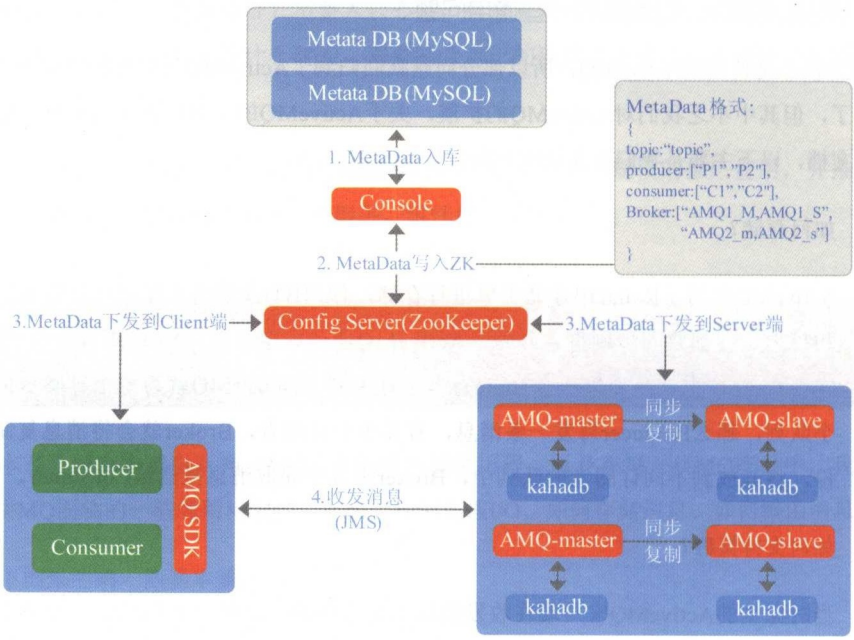


图4-7　客户端服务端整体架构

（2）加权随机发送。这种方式下我们可以对一个Topic分配的多个Broker设置不同的权重，发送时，Producer就根据权重来进行选择，权重越高被选中的几率越大。这种方式有个好处就是当一个Broker有异常时我们可以将其权重置为0，这样Producer就不会往上发消息，待其恢复之后我们再逐步将权重给予恢复，这就能够避免在单个Broker出异常时客户端大量报错的情况。

消费路由：Broker随机选择一个当前在线的Consumer并将消息推送给它，如图4-8所示。

4．如何处理消费失败的消息？

原生ActiveMQ的Consumer在成功处理一条消息之后会向Broker发一个ACK，以确认消息是否被成功消费。当处理一条消息失败之后，就会向Broker返回一个消息处理失败的应答，此时，当Broker收到这条处理失败的应答时，就会将这条处理失败的消息放到“死信队列”（DLQ）中。针对每个普通队列，Broker端都对应着这样一个死信队列，此队列的消息不能被消费者所获取。对此，我们对客户端进行了扩展，在消费出错时拦截错误信息，然后将出错的消息通过SAF协议（京东的SOA框架）调用“重

试服务”将异常消息入库，入库成功之后，Consumer再向Broker发送一个消费成功的ACK，这时Broker就会将此消息删除，实际上这样就将Broker端的消息转移到了库里。而后，Consumer再根据一定的策略通过SAF去调用“重试服务”获取之前入库的消息进行处理。处理消费失败消息的整体流程如图4-9所示。

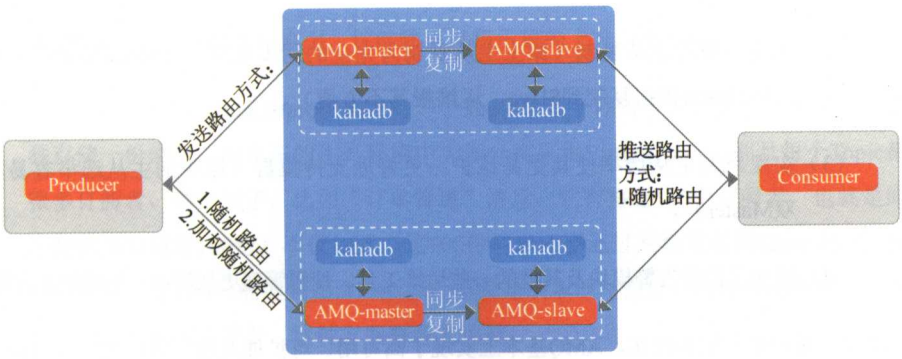


图4-8 客户端路由模型

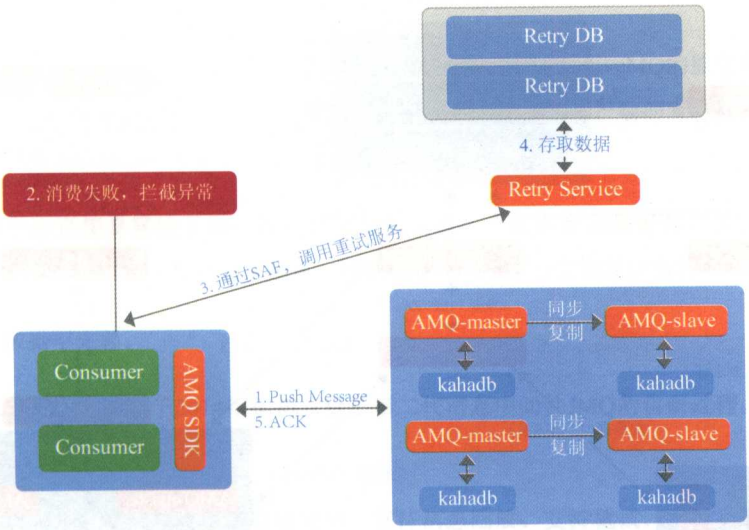


图4-9 处理消费失败消息的整体流程

5. 如何生成消息轨迹？

原生ActiveMQ并不支持消息轨迹，我们扩展了服务端，对写消息和消费消息进行了拦截，在消息入队成功和消费成功后分别记下日志，再由Agent采集日志并进行归档，Agent将元信息写入归档库，然后将消息内容写入JFS（JD File System，京东的分布式文件系

统）。归档成功后可在管理控制平台追踪消息的处理轨迹，必要时还可以下载消息体。

6. 其他扩展、优化。

（1）优化了Broker写消息逻辑，将性能提升到单实例TPS 4000+（原生单实例TPS 3000+）。

（2）实现了新的主从复制（原生主从复制为完全的同步复制，在Slave落后时需要手动将Master数据复制到Slave，运维极其不方便）。

（3）实现新的主从选举使其更易维护，更好地支持集群（原生的主从选举容易出现双Master）。

（4）增加了监控告警模块及其他的一些运维工具，使得管理更加简单，运维更加方便。

经过一系列的扩展和优化，AMQ基本上实现了高可用、可扩展及统一管理消息等目标，初步形成了一套完整的企业级消息中间件解决方案。AMQ平台粗略的整体架构图如图4-10所示。

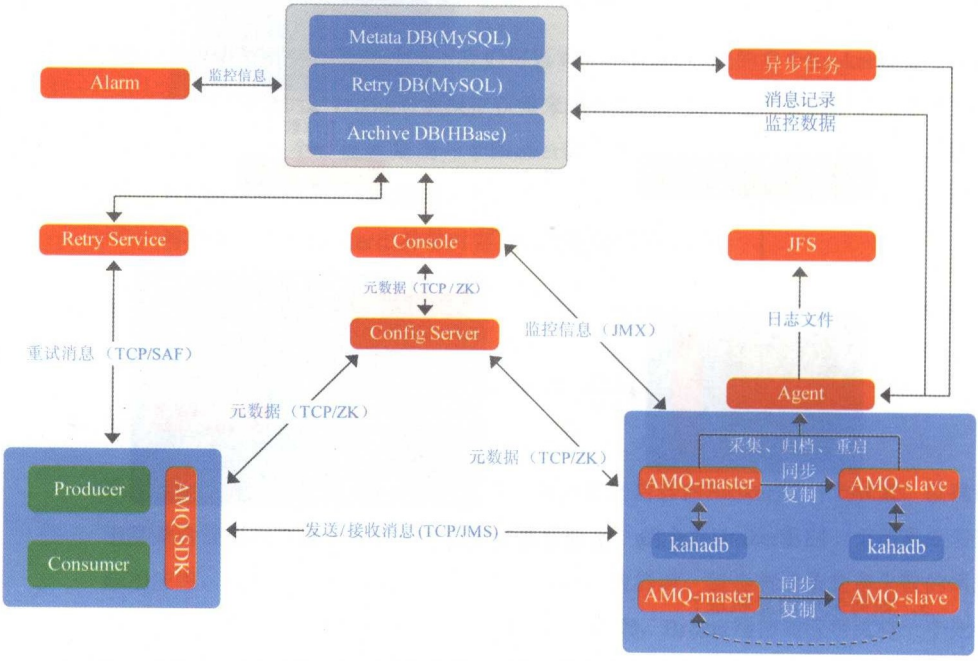


图4-10　AMQ整体架构

4.2.2 自主研发JMQ

到2014年年初，AMQ几经优化已经很成熟了。但随着公司业务的发展，接入的主题数量越来越多，消息量也成倍增长。此外，随着公司在各地新建机房，应用的部署结构也变得比较复杂，这就有了跨机房部署等需求。加上AMQ本身的一些问题，归结原因如下。

◎Broker较重，采用B-Tree索引，性能随消息积压量的上升急剧下降。

◎Broker端采用的VirtualTopic模式针对一个Topic有多个订阅者的情况会对每个订阅者单独存储一份消息。而京东的生产环境中大部分都采用VirtualTopic，并且每个Topic都有很多订阅者，举个例子，比如“订单管道”消息，它有将近100个订阅者，也就是同一个数据要写将近100份，不仅如此，这100份消息还要通过网络发送到Slave上，经过这些流程，写入TPS只能达到几百。所以，不管是从本地写性能、网络利用率来看，还是从存储空间利用率来看，这种方案都急需调整。

◎Broker逻辑复杂，其模型就决定了无法在其基础上扩展消息回放、顺序消息、广播消息等个性化需求，而实际使用过程中又比较渴望支持此类特性。

◎重客户端，由于集群、异常消息重试等功能都是通过扩展ActiveMQ的原生客户端并引入SAF、ZooKeeper等服务得以支持的，一定程度上增加了客户端的复杂度，相应地，在客户端的稳定性、可维护性等方面就打了折扣。

◎注册中心直接暴露给了客户端，这样最明显的一个缺点就是随着客户端实例数的增多，注册中心的连接数也越来越多，这就很难对注册中心实施保护措施。

◎监控数据不完善。

基于以上原因，我们于2014年年初开启了第二代消息中间件JMQ的自研过程。主要做了以下一些工作。

◎JMQ服务端：实现轻量级的存储模型、支持消息回放、支持重试、支持消息轨迹、支持顺序消息、支持广播消息等，并兼容AMQ客户端使用的OpenWire协议。

◎JMQ客户端：实现轻量级客户端只和Broker通信，支持动态接收参数，内置性能采集，支持跨机房。

◎管理控制平台：管理监控功能更加强大。

◎HTTP代理：基于Netty，支持跨语言。

JMQ如何解决MQ面临的一些通用问题

1．如何解决IO问题？

JMQ没有采用AMQ通过自己开发重复造轮子的方式解决IO问题，而是使用了Netty 4.0，此框架开源，支持epoll，编程模型相对简单。这在一定程度上减少了服务端的开发工作，也降低了服务端的复杂度。

在应用层，我们自定义了JMQ协议，序列化和反序列化也完全自己开发。这种序列化、反序列化的方式虽然在一定程度上降低了我们的开发效率，但我们不用考虑采用第三方的序列化和反序列化方案会带来的性能损耗问题，对于性能上的提升是显而易见的。

2．如何存储消息？

JMQ存储分为日志文件（journal）、消息队列文件（queue）及消费位置文件（offset），这三类文件都存储在Broker所在机器的本地磁盘上。

日志文件，主要存储消息内容，包括消息所在队列文件的位置，其特点如下。

（1）同一Broker上不同Topic的消息顺序追加到同一日志文件上，日志文件按固定大小切分。

（2）文件名为起始全局偏移量。

（3）日志文件同步刷盘。

由于JMQ主要使用在可靠性要求极高的下单、支付等环节，所以Broker必须保证收到的每条消息都落到物理磁盘上，设计这样一种日志文件主要是为了提高多Topic大并发下的磁盘写性能。不仅限于模型的设计，为了提高写性能，我们还在逻辑上实现了Group Commit。图4-11为JMQ中Group Commit的基本示意图。

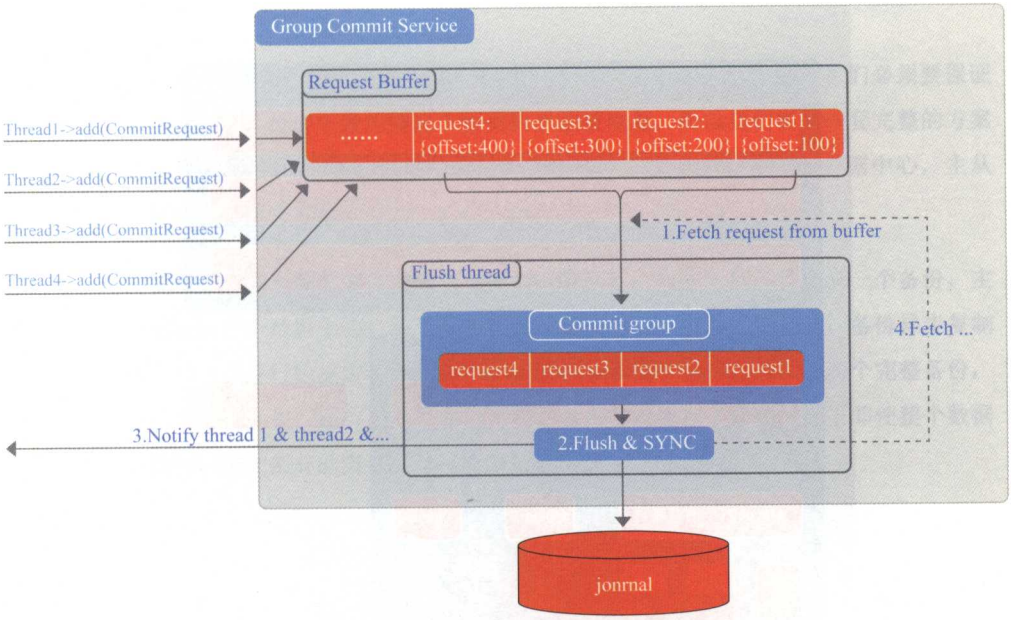


图4-11 Group Commit示意图

消息队列文件，主要存储消息所在日志文件的全局偏移量，此文件有以下特点。

（1）同一Broker上的不同Topic队列信息存储在不同的队列文件上，队列文件按固定大小切分。

（2）文件名为全局偏移量。

（3）顺序追加。

（4）队列文件异步刷盘。

由于在日志的写入是单线程的，所以在写入之前就可以提前获取到消息在队列文件中的位置，并将这个位置写入日志文件中。所以，只要日志文件写成功，即便队列文件写失败，我们也能从日志文件中将队列恢复出来，因此队列文件采用异步刷盘的方式。日志和队列文件结构如图4-12所示。

消费位置文件，主要用于存储不同订阅者针对某个Topic所消费到的队列的一个偏移量。

图4-13简单描述了消息在服务端的流转过程。

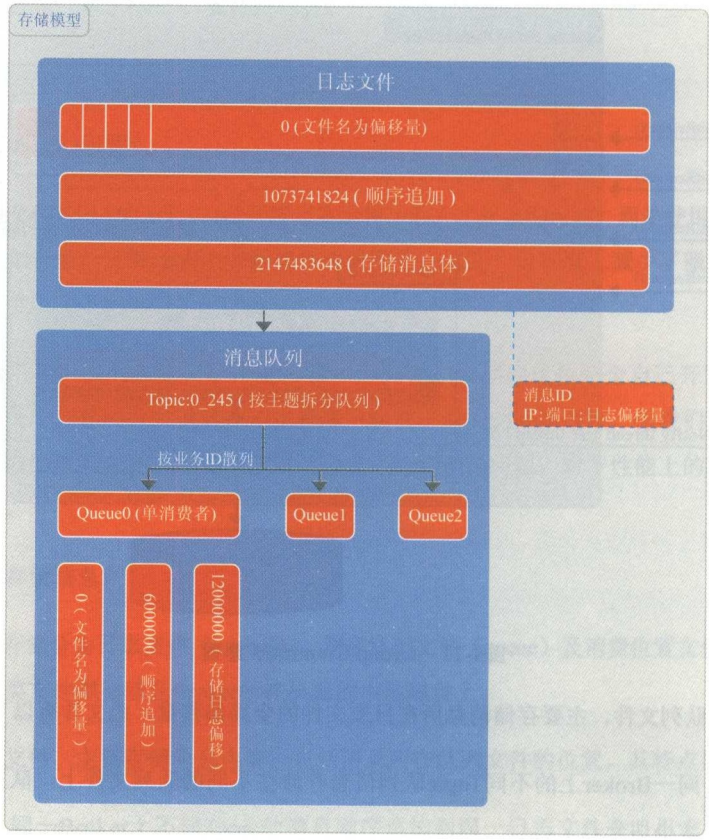


图4-12　日志文件和队列文件模型

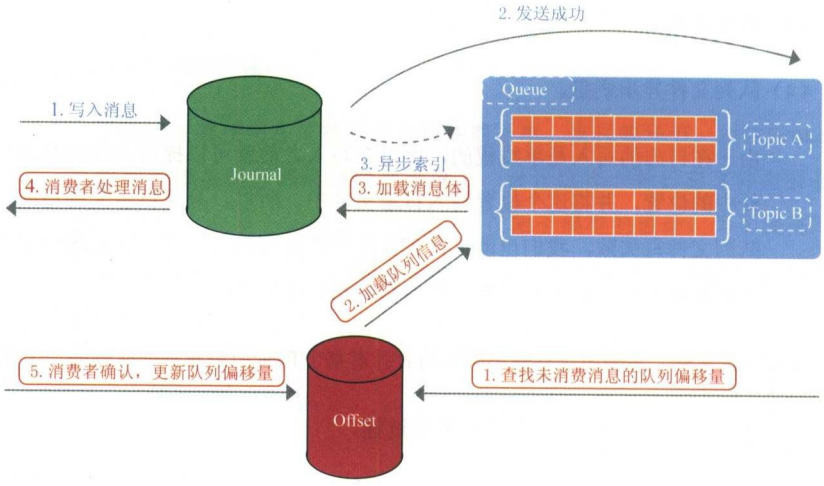


图4-13　消息流转示意图

3．如何容灾？

说到容灾，前面谈到每条消息在Broker上都会落地，但这远远不够，我们必须要保证在单机失效甚至机房失效的情况下数据依然能被恢复，所以我们需要一套完整的方案来进行数据灾备。在AMQ里采用的是“一主一从，主从分布在一个数据中心，主从同步复制”这样的方案。

JMQ实现了一套全新的复制方案（如图4-14所示）：一主一从，且至少一个备份，主从分布在同一个数据中心、备份分布在其他数据中心，主从同步复制、备份异步复制这样一种方式进行数据灾备。主从同步复制保证了同一条消息至少有两个完整备份，即便一个备份丢失，还有另一个备份可用。备份节点保证了极端情况下即使整个数据中心挂掉，绝大部分的消息还能得以恢复。

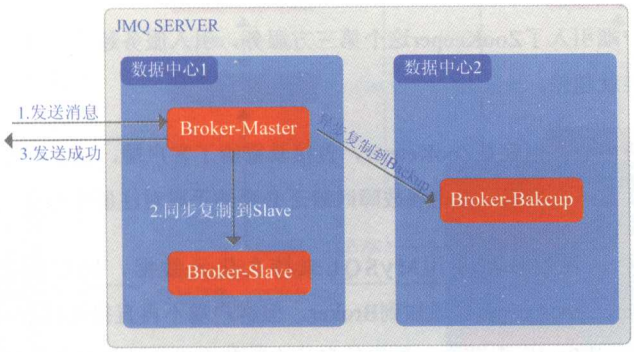


图4-14　JMQ复制方案

4．push 还是pull？消息如何路由？

JMQ采用pull模式，也就是消息由Producer发送到Broker之后是由Consumer主动发起请求去Broker上取消息的。

发送者路由和AMQ客户端采取的策略基本相同。

消费者路由和AMQ差不多也是随机，不同点在于JMQ是pull模式，在Broker端采取长轮询策略。

5．如何处理消费失败的消息？

与AMQ不同，由于JMQ 的Broker直接支持重试，因此Consumer在处理消息失败时会直接向服务端发送一个重试消息命令，服务端接到命令后将此消息入库。随后在

Consumer发起拉取消息命令时，服务端再根据一定的策略从库里将消息取出，返回给Consumer进行处理。这样做带来的一个好处就是JMQ客户端减少了一个第三方服务依赖。

6. 如何记录消息轨迹？

JMQ消息轨迹功能的整体流程和AMQ基本相同，主要不同点在于JMQ将消息轨迹相关信息存储到了HBase上，这样，JMQ消息轨迹信息的存储周期就会变得更长，可以存储的量也更大了，并且采用多种手段优化之后性能也得到了极大的提升。

7. 如何管理元数据？

通过第一部分我们知道，AMQ的元数据会持久化在MySQL中，并在入库的同时写入ZooKeeper，再由ZooKeeper下发到Broker和客户端。这样就有如下2个问题。

（1）客户端引入了ZooKeeper这个第三方服务，引入服务越多，客户端稳定性和可维护性就越差。

（2）注册中心，也就是ZooKeeper，直接暴露给了客户端，这样就会导致注册中心的连接数越来越多，在出现故障时缺乏必要的手段对注册中心进行保护。

在JMQ中，我们依然利用MySQL来持久化元数据，同时也会将元数据写入ZooKeeper，ZooKeeper再通知到Broker，但客户端不再直接连接ZooKeeper，而是连接Broker，从Broker上获取元数据信息。由于每个Broker都有全量的元数据信息，所以客户端连接任意的Broker都能获取到元数据信息。这种设计就带来了如下3个好处。

（1）减少了客户端对ZooKeeper服务的依赖，至此，客户端就只需和Broker通信，客户逻辑得到了简化，客户端的稳定性得到了极大提升。

（2）ZooKeeper不再暴露给客户端，这样ZooKeeper的稳定性也就有了保证。

（3）由于连接任意一个Broker都能获取到元数据，所以在极端情况下即便有个别Broker宕机也不影响客户端获取元数据，所以从另外一个角度来看，这又提高了我们注册中心的可用性。

8. 除了以上提到的一些基本问题之外，我们还解决了很多技术问题，由于篇幅原因就不在此一一罗列说明，其中包括但不限于如下4项。目前，JMQ架构图如图4-15所示。

（1）如何实现严格顺序消息。

（2）如何实现广播消息。

（3）如何实现两阶段事务。

（4）如何实现消息回放。

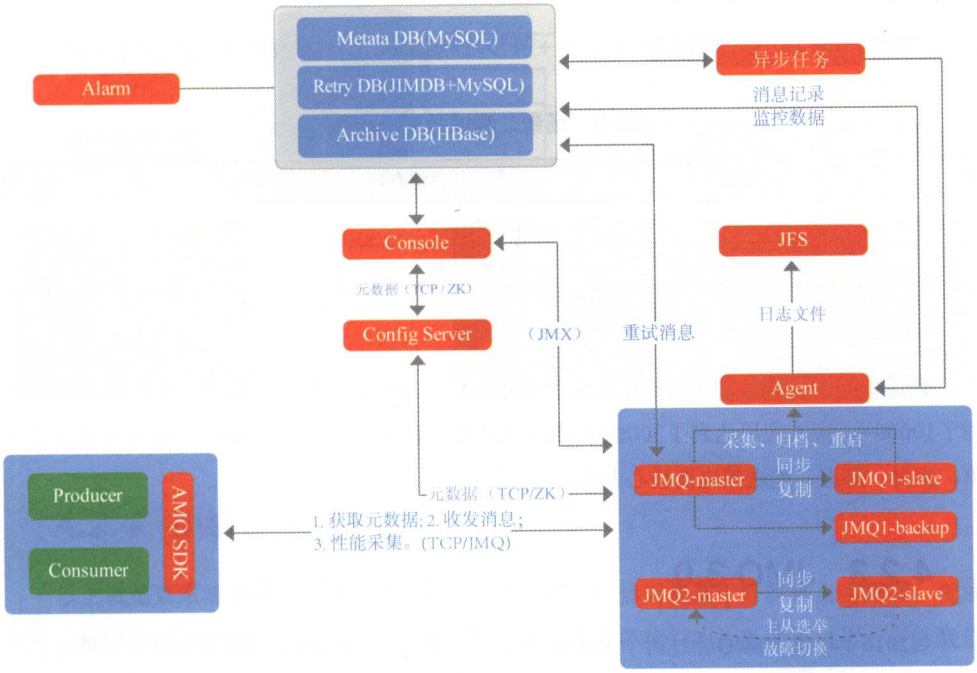


图4-15 JMQ架构图

JMQ性能数据

测试所用机器的情况如表4-3所示。

表4-3 测试机配置

操作系统	CPU	内存	磁盘	机器数量
CentOS release7	Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz	64GB	TOSHIBA AL13SEB300 300GB*8 Raid5	3台机器 1台部署客户端 2台部署Broker

场景一：一主一从，Master同步刷盘 + 同步复制到Slave，相关参数和测试结果如表4-4所示。

表4-4 场景一测试数据及结果

线程数	消息量	TPS
30	10000029	25000+
100	10000099	42000+

场景二：一主一从，Master异步刷盘 + 异步复制Slave，相关参数和测试结果如表4-5所示。

表4-5 场景二测试数据及结果

线程数	消息量	TPS
30	10000029	43000+
100	10000099	140000+

JMQ规模

经过JMQ开发团队的不懈努力，到2016年年中，JMQ已经初具规模，接入的Topic数量达到了10000+，接入应用达到了3000+，单日消息入队数量已经超过了500亿。接入者涵盖了下单、支付、库存、配送等电商核心业务。

4.2.3 JMQ 3.0

到2016年年中，JMQ经过两个大版本的迭代，线上运行的JMQ 2.0版本已经非常稳定了，性能也很优秀，但其发展并未止步。因为随着公司业务的快速发展，JMQ服务的规模也在迅速扩充，那么如何构建一个多样化的平台来适应业务的新需求，以及如何提升系统的可维护性就显得尤为重要，所以在2016年下半年我们设计并实现了JMQ的第三个版本——JMQ 3.0。JMQ 3.0有如下6个重要的特性。

◎优化JMQ协议。

◎优化复制模型。

◎实现Kafka协议兼容。

◎实现全局负载均衡。

◎实现全新的选举方案。

◎实现资源的弹性调度。

JMQ 3.0在功能上和架构上都会有一个大的升级，而且从线上运行效果来看，新版本在性能上又有了一定的提高。JMQ 3.0的整体架构如图4-16所示。

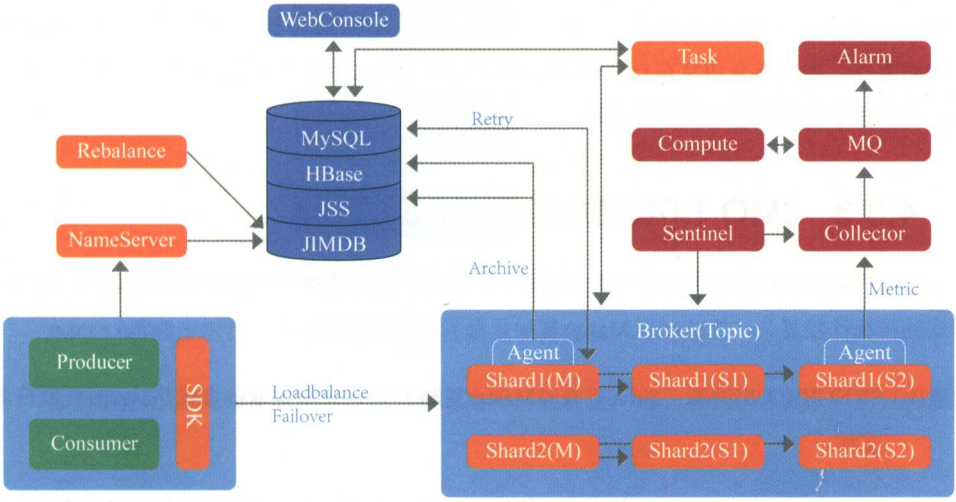


图4-16 JMQ 3.0整体架构

4.2.4 小结

以上就是京东消息中间件的演进过程。它从无到有，然后从基于开源到自主研发，发展至今，我们还在不断地探索如何让它更加完善，更好地支持我们的业务发展。下一步，我们也希望它能回到开源的怀抱，为开源尽绵薄之力。

4.3 JMQ复制技术解析

4.3.1 简介

本节介绍京东在自研消息队列平台时在保证数据可靠性的方式上进行的探索和思考，讲述了旧版中日志复制的实现方式及新版对其改进的要点，希望以此抛砖引玉与同行切磋成长。

4.3.2 引言

消息队列（MQ）作为应用系统之间的解藕机制、流量洪峰的缓冲池，已被各大互联网公司广泛采用。JMQ是京东完全自主研发的消息中间件，它有效地支撑了618和双11大流量的考验。随着京东内部应用系统的不断完善，目前已经有差不多3006个应用接入了JMQ。京东的黄金流程早已全面JMQ化。现在，互联网公司普遍采用廉价服务器 + 多份复本的方式保证数据的可靠性。本节将介绍京东自主研发的JMQ 1.0和JMQ 2.0两个版本系统在主从复制方案的设计和实践中的不同思考和解决方案。

4.3.3 JMQ 1.0实现方案

JMQ 1.0在架构设计时优先考虑低延迟和高吞吐，因此选择一主多从的复制模型。但当时考虑的情况比较复杂，把Slave从Master复制分为了如下3个阶段。

1. SYNCHRONIZING：Salve正在与Master复制，Slave使用拉取的方式从Master复制日志文件。

2. INCREMENTAL：Slave与Master的日志差距小于某阈值。处于这种状态时，将Master新收到的消息推到Slave后被保存在临时文件中，Slave同时还不断从Master拉取剩余日志数据。当Slave从Master把数据全部复制完成后，合并临时文件到正常的日志文件，同时把状态更改成ONLINE。

3. ONLINE：Slave与Master日志已经完全同步，这时，Master如果收到新的消息，则会推到Slave，收到Slave返回的确认才向生产者发送ACK包（重要消息）。

以上几种状态的变迁如图4-17所示。

实际上，INCREMENTAL状态加入的主要目的是为了减少Master对外提供服务的恢复时间，只有Slave处于INCREMENTAL或ONLINE时，Master才是可写的。这也导致复制模型变得更复杂易错。

JMQ 1.0复制支持ASYNCHRONOUS（拉）和SYNCHRONOUS（推）2种模式，如下。

◎拉模式：由Slave向Master发送GetJournal请求，Master返回日志数据。

◎推模式：由Master向Slave发送包含日志数据的UpdateJournal请求，Slave返回成功/失败。在UpdateJournal请求超时后，Master会要求变更成拉模式。

图4-17 状态变迁

JMQ 1.0在线上运行了近两年时间，也陆续发现存在以下问题。

◎偶尔会出现Slave不能复制的问题，重启后问题解决。由于复制状态复杂，所以问题定位不易。

◎服务器收到写请求后会在处理线程时使用CountDownLatch等待复制成功向客户返回ACK，当客户端连接比较多且消息比较频繁时会造成处理线程不能充分利用。

◎重要消息要求Slave复制成功后才由Master向生产者发送ACK，系统实现中使用推模式确认Slave是否收到复制数据。如果需求改为重要消息有多份副本，那么至少有两份以上才认为写成功。如何处理“当两个Salve进行推操作时，如果其中一个出现故障，如何用其他的替代？故障Slave恢复后怎么办？”等问题。在未来计划实施的两地三中心战略中需要至少两个Slave中有一个成功复制来保证数据的高可靠性，基于目前的架构实现改造难度非常大。

4.3.4 复制的分类

在新JMQ复制协议设计之初，我们充分研究了目前主流的复制方式，总结如下。

◎多主Paxos：即网络中存在多个主节点都可以接受写操作，它们之间使用Paxos算法保证最终一致性。优点是多点可写，但缺点也很明显（如图4-18所示），在无冲突情况下也需要经历两阶段才能达成一致。

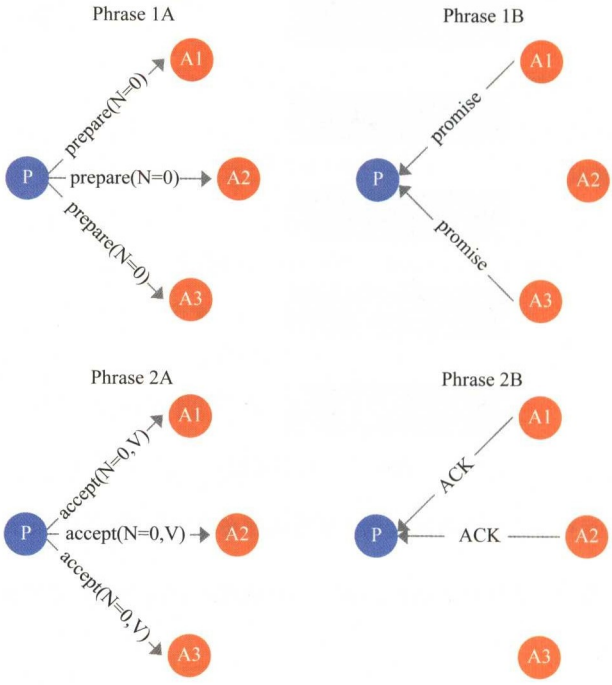


图4-18　多主Paxos

◎一主多从并联：分片由一个主节点和多个从节点构成，每个从节点直接从主节点复制数据。这种结构的优点是多从节点之间没有依赖关系、复制延迟低，缺点则是加大了主节点的压力且在主节点失效后选主也较麻烦。如图4-19所示。

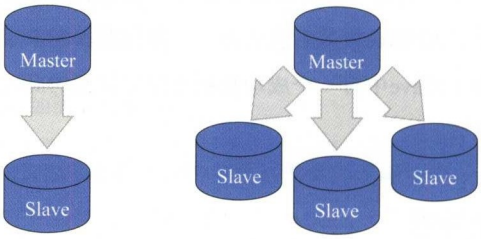


图4-19　一主多从并联

◎一主多从串连：分片由一个主节点和多个从节点组成一个以主节点为首的链式结构，每个节点都从上级节点复制数据。这种结构的优点是很好地解决了主节点失效后的选主问题（直接用下级节点替代），而缺点则是复制延迟较大（主节点到末尾节点）。如图4-20所示。

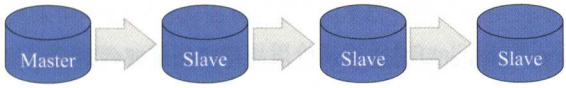


图4-20 一主多从串连

◎Raft复制：Raft复制实际上由如下两部分组成。

○Raft选主过程，目的是在多个Raft节点中通过投票的形式选择一个为Leader节点。

○日志复制过程，所有写操作都写入Leader节点并记入日志，多个Follower通过复制Leader日志保证数据可靠。

4.3.5 JMQ 2.0复制协议实现方案

正所谓“大道至简，知易行难”，JMQ 2.0复制架构的设计摒弃了之前使用的复杂模型。新架构中只使用拉模式由Slave从Master不断拉取日志数据保存到本地磁盘中。Slave与Master的同步状态简化为inSync和Online状态，根据Master与Slave之间的日志差距来判断Slave是否处于inSync状态（差距小于某个阈值就表示已经同步了）。同时，在Master上维护一个waterMark（水位）变量，它的含意是至少*N*个Slave已经同步到的日志位置。这里，*N*是同步服务器数，它是可配置的。在JMQ中的配置为1，因为JMQ分片典型配置由一主两从构成。水位概念参考图如图4-21所示。

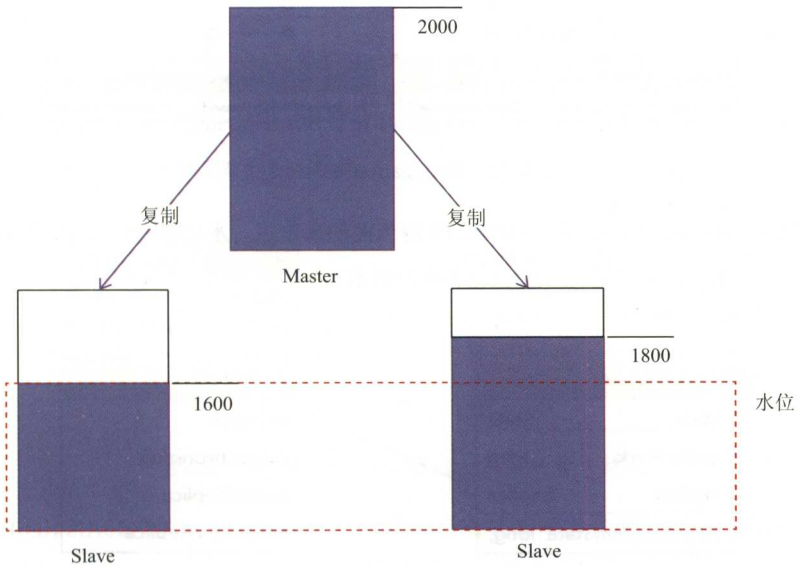


图4-21 水位概念参考图

图4-21红框中的水位是同步服务器数为2时计算出的水位值。如果同步服务器数为1，则根据水位算法应为1800。Master根据这个水位值就可以知道哪些消息已经被复制到Slave上，因此可以向生产者发送ACK确认包。

对于JMQ 2.0的实现，Master部分抽象了ReplicationMaster和Replica接口，ReplicationMaster的定义如图4-22所示。

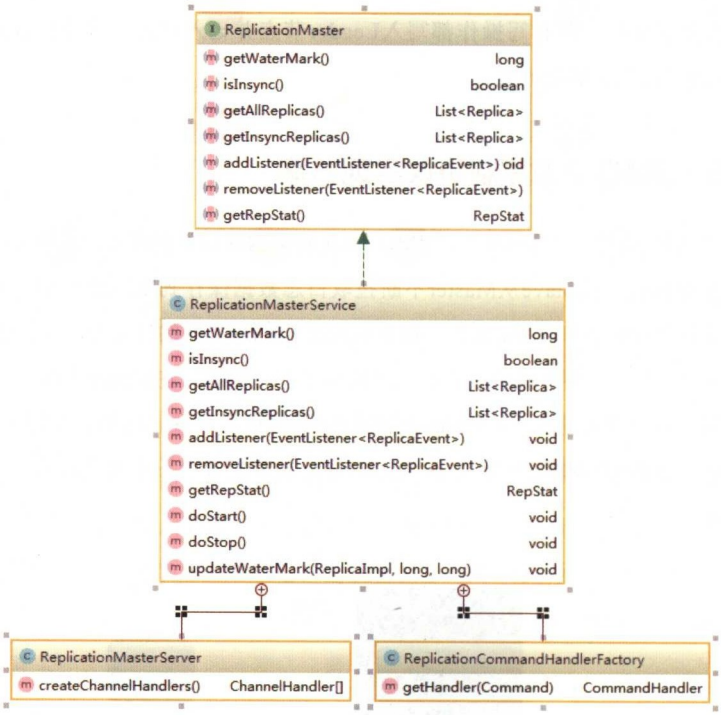


图4-22　ReplicationMaster定义

ReplicationMaster通过getWaterMark()获得当前的水位值。水位值变化时也向Listener发布ReplicaEvent事件。ReplicaEvent的定义如图4-23所示。

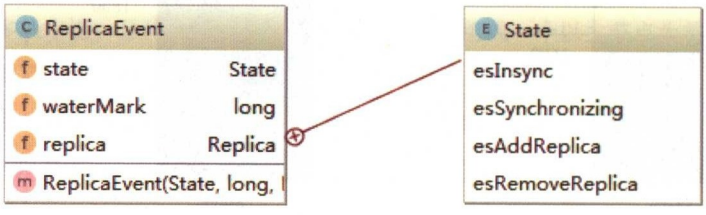


图4-23　ReplicaEvent定义

其中，ReplicationMasterService是实现类，它的作用是建立一个TCP listen服务，等待Slave连接。每个连接上来的Slave经过鉴权后都会创建一个Replica对象，如图4-24所示。

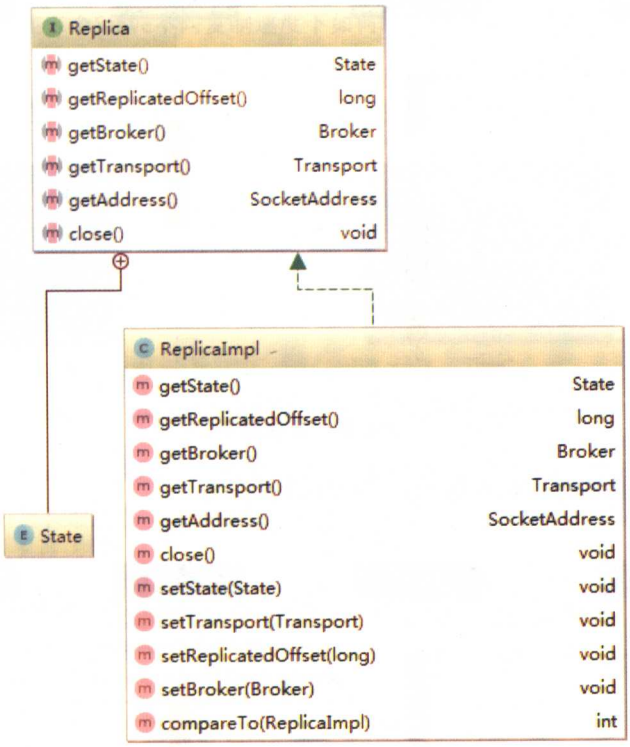


图4-24 Replica对象

所有的Replica通过发送GetJournal命令请求日志数据，而ReplicationMaster通过GetJournalAck返回对应数据块。其命令格式如图4-25所示。

图4-25 GetJournal和GetJournalAck的命令格式

GetJournalAck中的insync表示这个Replica是否处于同步状态（同步状态的Replica可以在Master失效时被选为主）。GetJournal还有一个隐含的作用是ReplicationSlave告诉

ReplicationMaster已经复制的数据位置。ReplicationMaster通过这个值来更新水位值，通过新的水位值可以判断哪些消息已经被成功复制到了Slave。具体时序可参考图4-26。

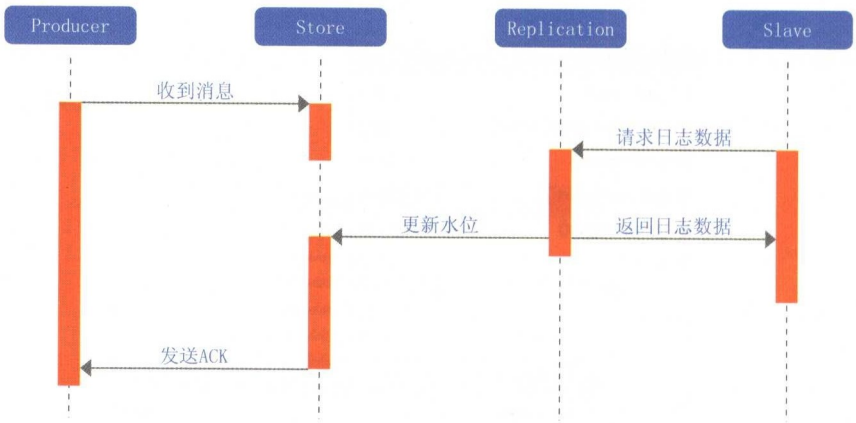


图4-26 具体时序

我们可以对比JMQ 1.0的复制时序，如图4-27所示。

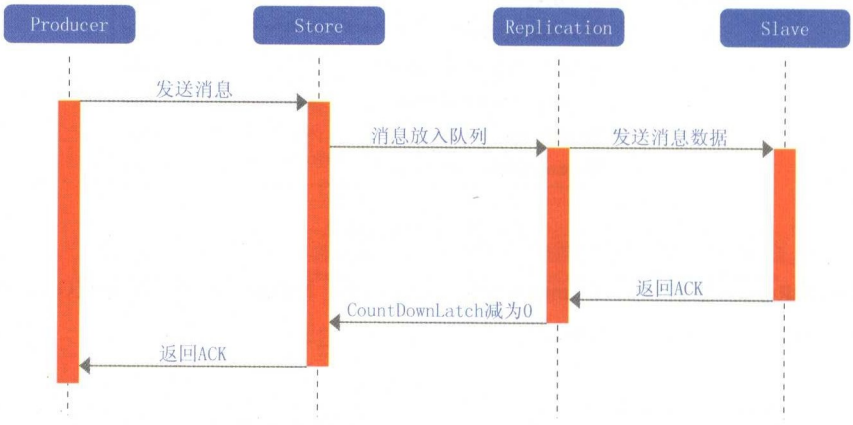


图4-27 JMQ 1.0的复制时序

不难发现，在收到Producer的消息请求后，JMQ 2.0复制的处理逻辑分成了两部分，分别在不同线程中完成存储和返回ACK，这样，处理线程得到了更充分的利用。并且返回给Slave的日志数据通常都是一大块数据，这也极大地提高了复制的吞吐量。而Slave对日志的请求本身也可作为上次复制数据正常到达的一种隐式确认。

在复制模块收到某一个Slave（从复制模块角度也可称为Replica）的确认ReplicatedOffset

后，可以把多个Replica以ReplicatedOffset为轴进行降序排序，然后从头部取InsyncCount（配置的同步服务器数）位置的Replica的ReplicatedOffset作为整个分片的最终水位值。

根据图4-28，可以计算出InsyncCount为2时，时刻Time1和Time2的水位分别为1700和1800。这就解决了在多个Slave不同复制速度下如何动态地确定已经成功复制的Offset值的问题。

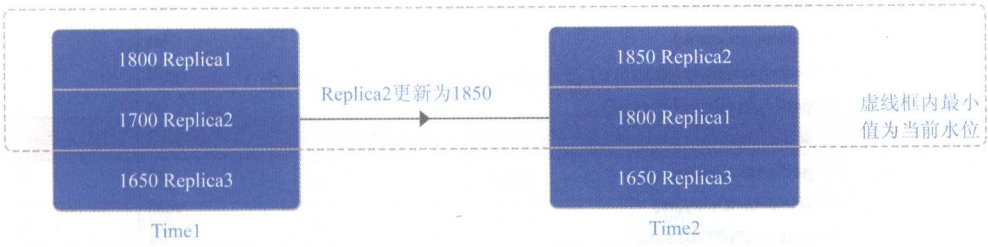


图4-28 InsyncCount为2时，时刻Time和Time2的水位

相对于ReplicationMaster被动地接收复制请求，ReplicationSlave则使用长轮询方式主动向ReplicationMaster发送拉取请求。ReplicationSlave的类定义如图4-29所示。

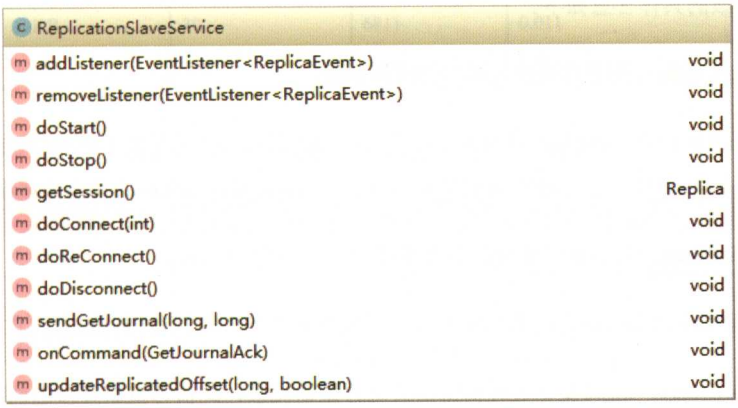


图4-29 ReplicationSlave的类定义

它的作用是不断从Master拉取日志数据，并存储到本地磁盘中。但要注意的是由于日志复制是单TCP连接大数据块传输，因此还要设置一下TCP的收/发缓存区大小才能达到较大的网络吞吐量（根据测试，1分钟左右即可，再大也不能显著提高性能）。如果在跨机房的复制中，最好还是做一下限流，防止在一个Master上挂了一台新Slave时进行疯狂复制而把带宽都跑满了的情况。

在复制模块初始化时，会传入ReplicationConfig对象，复制相关的主要参数都通过它进行设置，如图4-30所示。

ReplicationConfig	
getServerConfig()	ServerConfig
setServerConfig(ServerConfig)	void
getInsyncSize()	long
setInsyncSize(long)	void
getInsyncCount()	int
setInsyncCount(int)	void
getReplicaDeadTimeout()	int
setReplicaDeadTimeout(int)	void
getRestartDelay()	int
setRestartDelay(int)	void
getLongPullTimeout()	int
setLongPullTimeout(int)	void
getBlockSize()	int
setBlockSize(int)	void

图4-30　ReplicationConfig对象的定义

上面参数的具体含意如下。

◎ServerConfig：配置监听IP、端口等网络参数。

◎InsyncSize：表示在Master和Slave日志差小于此值时认为主从是处于Insync状态的（在以后介绍主从选举中会使用到Insync状态，只有处于Insync状态的Slave才有机会被选为主）。

◎InsyncCount：表示最低要复制多少份才向生产者返回ACK。

◎ReplicaDeadTimeout：用来判断多长时间没有网络传输就认为Replica已死亡。

◎RestartDelay：Slave复制过程出错后会进行重新连接，此值为重新连接的间隔时间。

◎LongPullTimeout：复制请求的长轮询等待时间。

◎BlockSize：每次复制数据块的大小（实际可能小于此设置值）。

4.3.6　测试结果

在京东线上典型服务器环境上进行测试，结果令人满意。具体测试环境如下。

◎机器配置：E5-2640 v3 (32core)，64GB RAM，SAS + 2TB Disk。

◎分片由一主一从组成。

◎生产者分别由1台和3台的2组机器分别测试。

◎测试方法为客户端每个线程同步发送消息，而服务端在写盘及复制成功后返回ACK。

生产或消费消息大小分别为1KB、10KB、100KB，结果如表4-6到表4-8所示。

表4-6 生产或消费消息大小为1KB时的结果

编号	消息大小	并发线程数	TPS（笔/秒）	平均响应时间（秒）	交易成功率
1	1KB	10	38824	0.001	100%
2	1KB	30	60673	0.001	100%
3	1KB	100	73322	0.003	100%

表4-7 生产或消费消息大小为10KB时的结果

编号	消息大小	并发线程数	TPS（笔/秒）	平均响应时间（秒）	交易成功率
1	10KB	10	7237	0.003	100%
2	10KB	30	6811	0.011	100%
3	10KB	100	6289	0.057	100%

表4-8 生产或消费消息大小为100KB时的结果

编号	消息大小	并发线程数	TPS（笔/秒）	平均响应时间（秒）	交易成功率
1	100KB	10	642	0.066	100%
2	100KB	30	669	0.146	100%
3	100KB	100	636	0.576	100%

4.3.7 总结

以上内容介绍了旧版和新版日志复制的原理。这也反映出在不同时期设计的重心差异，旧版的设计重心在于分片能尽量减少恢复时间，而新版则致力于简化处理逻辑并提升复制吞吐量。

4.4 CallGraph：分布式服务跟踪系统

4.4.1 CallGraph的产生背景

随着京东业务的高速增长，京东研发体系陆续实施了SOA化和微服务战略，以应对日益复杂的业务和急剧增加的应用种类。这些分布式应用彼此依赖，并通过共同协作来完成所有京东的业务场景。应用动态变化的复杂性和数量已超出想象，对其进行监控并试图掌控全局已非人力所及，迫切需要一种软件工具来帮助相关人员理解系统行为，从而为流程优化、架构优化、程序优化，以及扩容、限流、降级等运维行为提供科学的客观依据。

CallGraph是根据Google为其基于日志的分布式跟踪系统Dapper发表的论文*Dapper, a Large-Scale Distrbuted Systems Tracing Infrastructure*，由京东基础架构部自主研发的分布式跟踪系统，目前已经上线。业界亦有类似系统，比如淘宝鹰眼、新浪WatchMan等。但是，CallGraph除了提供与这些系统类似的功能外，还有其自身的特色，下面将作详细讲解。

4.4.2 CallGraph的核心概念

“调用链”是CallGraph最重要的概念，一条调用链包含了从源头请求（比如前端网页请求、无线客户端请求等）到最后底层系统（比如数据库、分布式缓存等）的所有中间环节，如图4-31所示。

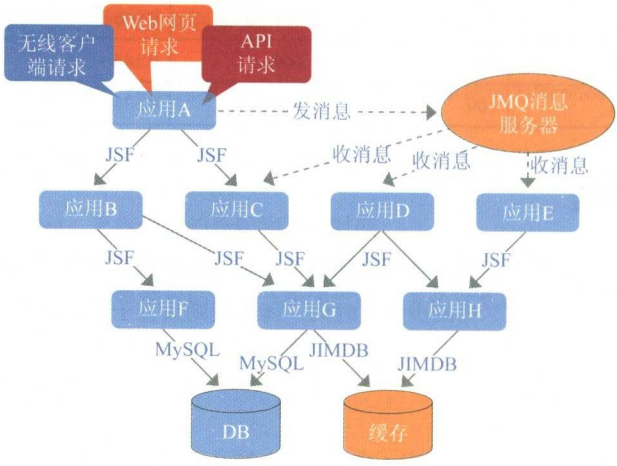


图4-31　应用间复杂的调用链

每次调用，都会在源头请求中产生一个全局唯一的ID（称为TraceId），通过网络依次将TraceId传到下一个环节，该过程被称为“透明数据传输”（简称“透传”）。图4-31中的每一个环节都会生成包含TraceId在内的日志信息，通过TraceId将散落在调用链中不同系统上的“孤立”日志联系在一起，然后通过日志分析，重组还原出更多有价值的信息。

4.4.3 CallGraph的特性及使用场景

CallGraph本质上是一种监控系统，但它提供了一般监控系统所没有的特性，每种特性都有其典型的使用场景，为相关人员提供了强大的问题排查手段和决策依据，现将对比总结于表4-9中。

表4-9 CallGraph与一般监控系统的对比

特性	一般监控	CallGraph	场景	一般监控	CallGraph
方法调用关系	只能提供有直接调用关系的方法调用信息	可以提供调用链上所有上下游方法的调用信息	单次调用的问题排查	需要查看N台机器的监控、业务日志，效率低下	应用日志关联TraceId，用TraceId查询调用链定位问题，关联链路上的异常堆栈和相关服务器的环境指标（CPU、IO等）
应用依赖关系	只能提供应用的直接依赖关系	利用调用链可以梳理出所有的应用依赖关系	容量规划	基本上缺乏系统、科学的方法，多数情况下靠以往经验或者片段信息进行决断	提供间接、异步依赖的调用量信息分析
			调用来源		提供直接、入口来源的分析
			依赖度量		提供强、高度及频繁依赖的分析
			调用耗时		提供TP分析、耗时瓶颈分析
			调用并行度		为链路并行、异步优化提供依据
			调用路由		根据调用量占比评价路由均衡性、热点分析
与业务数据集成	不能提供	可在调用链上“透传”业务数据	将公司业务与第三方业务进行关联	无手段	透传特定业务标识数据，提供“泛京东调用链”分析

4.4.4 CallGraph的设计目标

◎低侵入性。

作为非业务系统，应当尽可能少侵入或者不侵入其他业务系统，保持对使用方的透明性，这样可以大大减少开发人员的负担和接入门槛。

◎低性能影响。

CallGraph通过对各中间件jar包进行改造，完成日志数据的产生和收集，我们称这种改造为“埋点”。由于埋点都发生在业务核心流程上，所以应该尽最大努力降低对业务系统造成的性能影响。

◎灵活的应用策略。

为了消除业务方因使用CallGraph对其自身产生影响的担忧，应该提供灵活的配置策略，以让业务方决定是否开启跟踪，以及收集数据的范围和粒度，并提供技术手段保障配置生效。

◎时效性（time-efficient）。

从数据的产生和收集，到数据的计算/处理，再到展现，都要求尽可能快速。

4.4.5 CallGraph系统架构

如图4-32所示，为CallGraph系统架构图，下面会对架构相关部分作详细讲解。

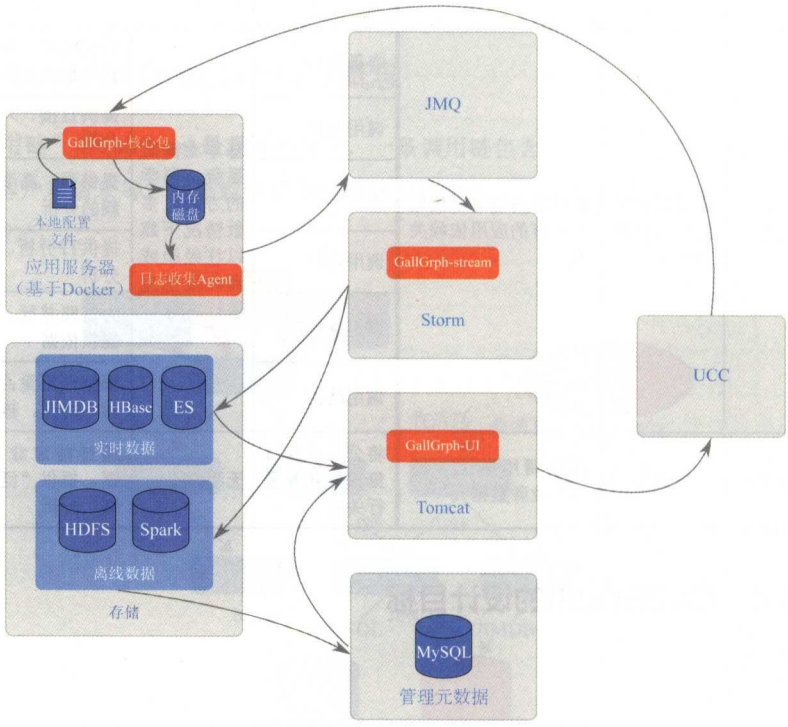


图4-32 CallGraph系统架构图

◎CallGraph核心包。CallGraph核心包被各中间件jar包引用，核心包里完成了具体的埋点逻辑，各中间件在合适的地方调用核心包提供的API来完成埋点；核心包产生的日志被存放在内存磁盘上，由日志收集Agent发送到JMQ里。

◎JMQ。JMQ是京东的分布式消息队列，利用其强劲的性能，充当日志数据管道，Storm将不断地消费里面的日志数据。

◎Storm。利用Storm进行流式计算，对日志数据并行进行整理和各种计算，并将结果分别存放到实时数据存储和离线数据存储中。

◎存储。包括实时数据和离线数据两部分存储。实时数据部分包括了JIMDB、HBase和ES，JIMDB是京东自己的分布式缓存系统，存放了调用量、TP等实时指标数据；利用HBase的SchemaLess特性，存放了固化后的链路数据，因为不同的链路包含的中间环节数量不一样，所以无法用像MySQL这样强Schema特性的存储，利用TraceId就可以从HBase里查询到某一次调用的所有中间环节的信息。离线数据部分包括HDFS和Spark，用于海量历史数据分析，并且还会把一些结果存放到MySQL中。

◎CallGraph-UI。这是CallGraph提供给用户的交互界面，在这里，用户可以查看属于自己的所有系统，以及各系统内应用的调用链路的详细情况，包括应用间的相互依赖关系图，某种服务方法的来源分析、入口分析、路径分析，以及某次具体的调用链路的详情等，还可以对应用进行诸如“采样率”等配置的设置。

◎UCC。UCC是京东自己的分布式配置系统，CallGraph用它来存放所有的配置信息，并且同步到应用服务器本地的配置文件中。核心包将定期检查这些配置文件，以使配置生效。当UCC故障后，也可以通过直接操纵本地配置文件，使配置生效。

◎管理元数据。存放CallGraph的管理元数据，比如链路签名与应用的映射关系、链路签名与服务方法的映射关系等。

4.4.6 CallGraph的技术实现

埋点和调用上下文透传

该部分属于架构图中的CallGraph核心包的重点部分，也是难点部分。CallGraph核心包完成埋点逻辑，如图4-33所示。

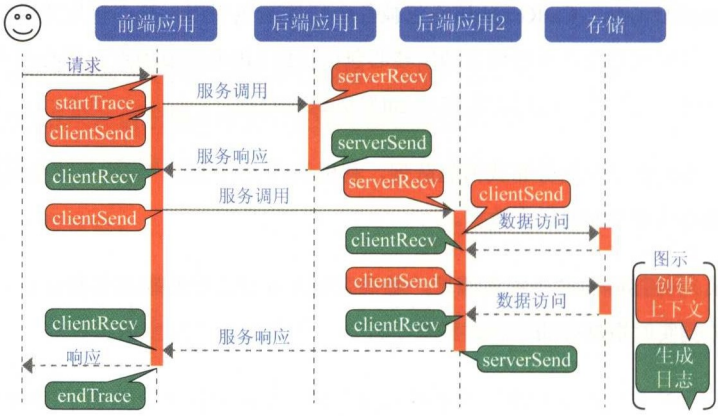


图4-33　埋点逻辑时序图

前端应用和各中间件jar包引入CallGraph核心包，前端应用利用Web容器的Filter机制调用核心包的startTrace开启跟踪，收到响应后调用endTrace结束此次跟踪，各中间件在合适的地方调用核心包提供的clientSend、serverRecv、serverSend和clientRecv等原语API。图4-33中，橙色部分代表完成“创建上下文”，绿色部分代表完成“生成日志”。

对于进程间的上下文透传，调用上下文放在本地ThreadLocal，对业务透明，调用上下文在中间件的网络请求中传递，在对端收到后进行重组并还原出调用上下文，过程如图4-34所示。

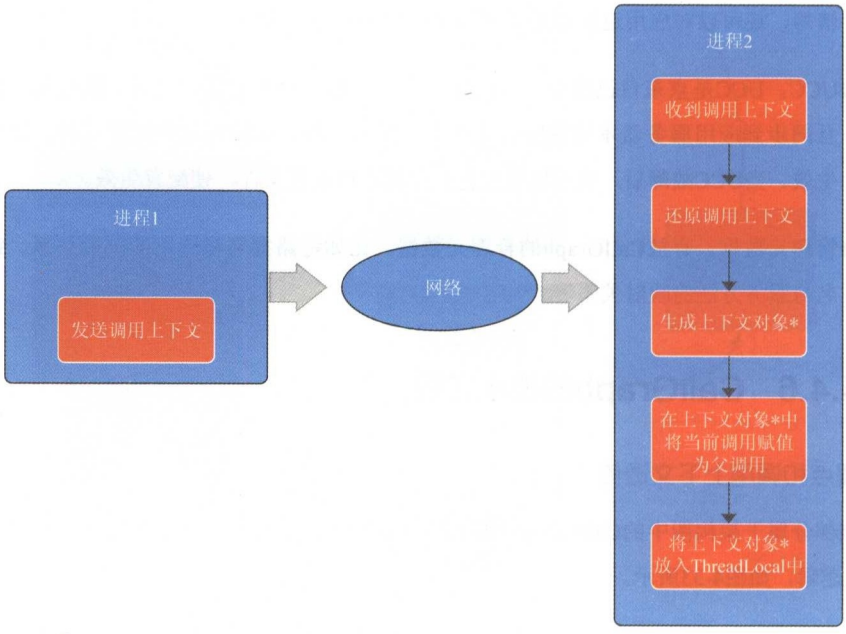


图4-34　调用上下文的传递

对于异步调用，将涉及线程间上下文透传，通过Java字节码增强的方式在CallGraph核心包载入期植入增强逻辑，以透明的方式完成线程间的上下文透传。这里又可以分为两种类型，一种是直接创建新线程的方式，如图4-35所示，这种方式通过对JDK线程对象（Thread）进行增强完成，子线程将父线程的上下文作为自己的上下文（即图中的“子上下文”）；另外一种情况涉及Java线程池中被缓存的线程，对于这种情况就不存在“父子”线程关系了，这时会通过对各种JDK线程池的增强，实现上下文透传，如图4-36所示。

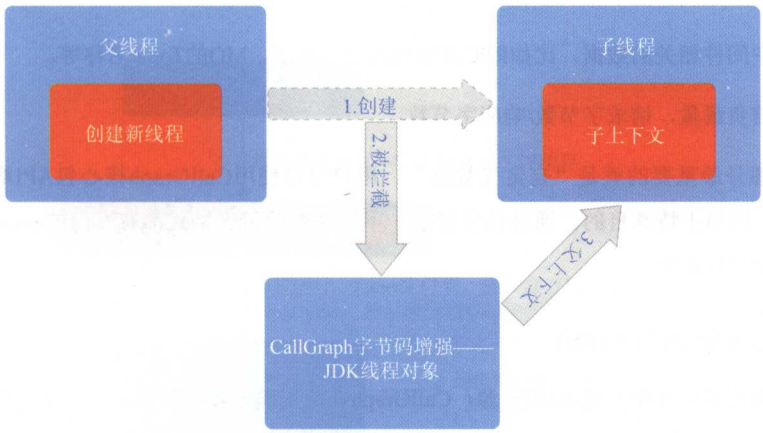


图4-35 父子线程间上下文的透传

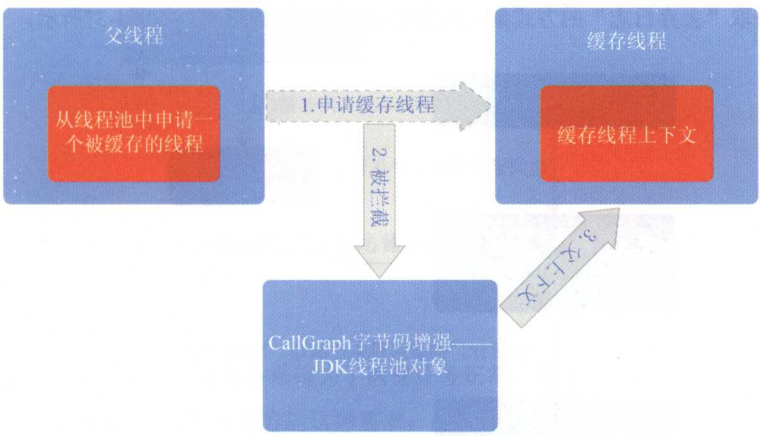


图4-36 Java线程池的上下文透传

上述过程对开发人员完全透明，对运维人员来说也很方便，做到了“低侵入性”。

日志格式设计

CallGraph的日志格式需要满足不同中间件的特定要求，同时还要保证版本的兼容性。总体上说，CallGraph的日志格式分成固定部分和可变部分，其中固定部分由如下五大部分组成。

◎相关参数信息：TraceId、RpcId、开始时间、调用类型、对端IP。

◎调用耗时。

◎调用结果。

◎与中间件相关的数据，比如RPC调用的接口和方法、MQ的Topic名称等。

◎通信负载量，请求字节数/响应字节数。

可变部分最重要的就是“自定义数据”，用户可以使用CallGraph核心包API增加自己的特殊字段，以用于特殊目的。通过抽象设计，不同场景的日志格式都有专门的encoder类，在输出日志时配套使用。

高性能的链路日志输出

为了彻底避免和业务竞争I/O资源，CallGraph在应用服务器上开辟了专门的内存区域，并虚拟成磁盘设备，核心包产生的日志存放在这样的内存磁盘上，完全不占用磁盘I/O，并且速度极快。同时开发专门的日志模块、日志输出采取批量、异步方式写入内存磁盘，并在日志量过大时采取“丢弃日志”的方式最大程度地降低对业务的影响，如图4-37所示。

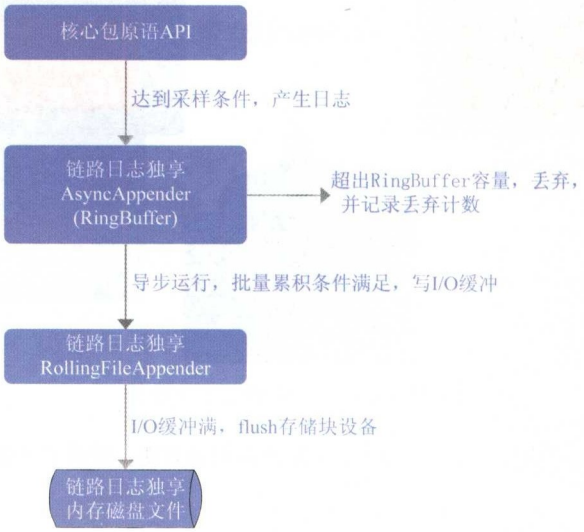


图4-37　链路日志输出的实现过程

TP日志和链路日志分离

为了最大程度地减少对业务性能的影响，在实践中，多数情况下会开启“采样率”机制，比如1000次调用，只收集1次调用的信息，这样可以极大地降低日志产生量。但是对于TP指标来说，必须记录每次调用的TP值，否则提供的TP50、TP99、TP999指标将不准确，从而使指标变得没有意义。从本质上说，链路信息和TP性能指标是两种不同属性的数据，因此在核心包里分别对这两种数据进行独立处理，彼此互不影响，采用各自的日志收集及输出策略。TP指标的处理如图4-38所示。

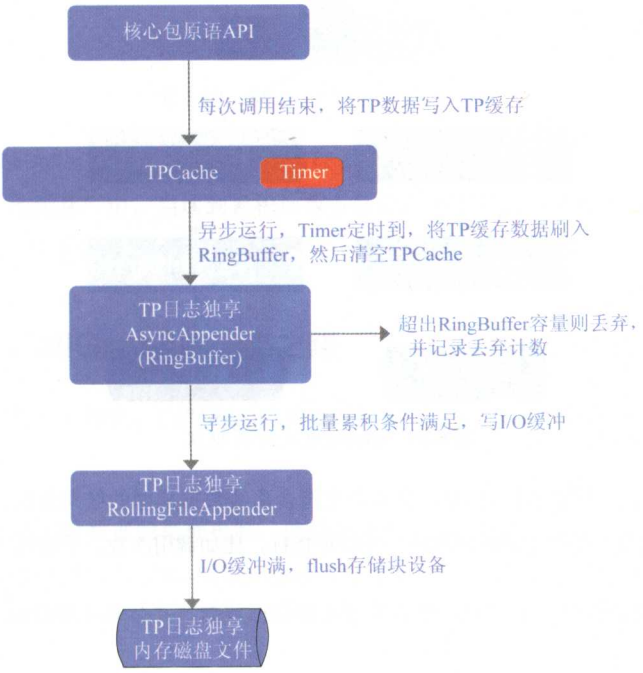


图4-38 TP日志输出的实现过程

实时配置

当双11或者618大促时，各业务系统为了确保业务正常，基本上都会对非业务系统采取降级的手段。CallGraph为了满足业务方的这种需求，提供了丰富的配置和降级手段。CallGraph提供了基于应用、应用分组、应用服务器IP等多维度的配置方式，每个维度上都提供了“是否开启链路跟踪”“链路采样率”“是否开启TP跟踪”“TP颗粒度”等配置项，以供业务方根据情况来使用。

业务方通过CallGraph UI管理端自助设置业务的各配置项。全部配置信息存放在UCC（京

东的分布式配置系统）上，同时也会同步到应用服务器的本地配置文件中。CallGraph核心包有专门的Daemon线程定期访问本地的这些配置文件，以使配置生效。当UCC出现故障，不能被正常访问时，也可以直接操纵这些本地配置文件，确保配置立即生效。

Storm流式计算

所有日志，不管是链路日志还是TP日志，最后都必须经过Storm进行计算产生结果数据，并分别存储到实时数据存储和离线数据存储中，如图4-39所示。

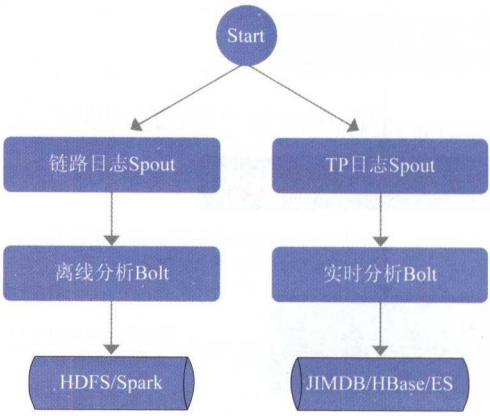


图4-39　Storm流式计算框架

离线分析Bolt分析链路日志信息，负责产生符合离线数据模型的结果数据，由Spark进行计算，得到大时间尺度下的固定后的链路的一些特征指标，比如调用次数、平均耗时、错误率等。

实时分析Bolt分析TP日志信息，负责生成实时指标数据，并存储在JIMDB中，供CallGraph UI调用展示。

实时数据分析——秒级监控

这是CallGraph区别于其他类似系统的一大功能。其他类似系统只提供链路日志分析，而链路日志的分析需要积累海量数据，然后借助大数据相关技术进行分析，实时性较低。针对业务方对实时分析的需求，CallGraph采用分布式缓存系统JIMDB来存放实时数据，针对来源分析、入口分析、链路分析等可以提供1小时内的实时分析结果（为JIMDB中的数据设置过期时间，可以自动过期），其中涉及调用量、调用量占比、TP性能指标等展示，该功能被内部称为“秒级监控”。“秒级监控”需要对TP日志进行分析，原理如图4-40所示。

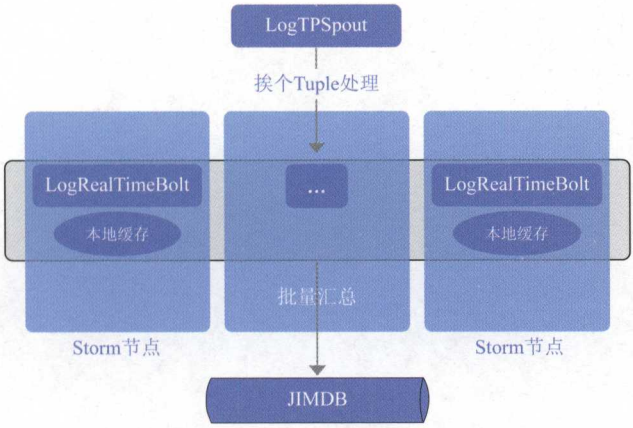


图4-40 Storm中的实时数据分析

LogRealTimeBolt将从LogTPSpout中得到TP原始日志，进行整理、分析和计算，并将结果暂时缓存在“本地缓存”中，当达到累积计数条件后，再批量地汇总到JIMDB存储中，这样做的好处是可以先在本地进行合并计算，另外也减少了JIMDB的I/O次数。

4.4.7 CallGraph的未来之路

与其他中间件产品相比，CallGraph的建设时间相对较短。CallGraph正在陆续完善及能够提供的功能如下。

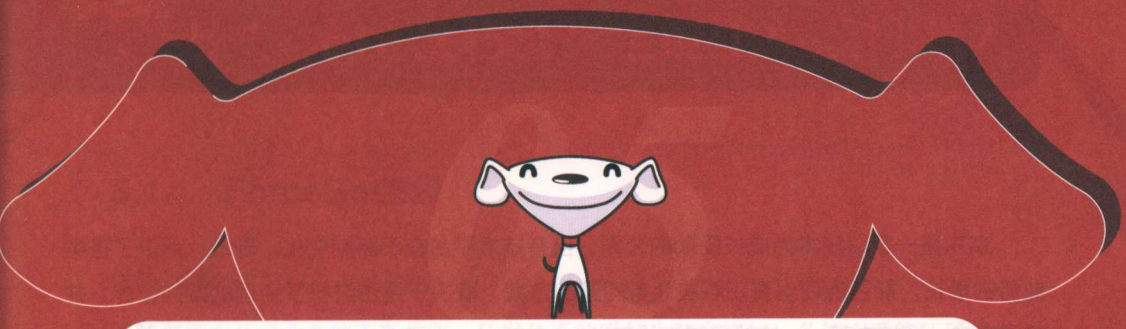
◎进一步优化实时数据的处理机制，使得时延更低，达到真正的“实时”。目前，该功能由于需要经过日志收集、JMQ及Storm等过程，所以存在十几秒到几十秒钟的时延，属于“准实时”的范畴。

◎完善实时的错误发现及报警机制，进一步提高发现问题的及时性。

◎接入所有中间件服务特别是JED与JMQ，进一步丰富调用链内容，使调用链更长更完整。

◎提供完整的API接口，将调用链数据共享给兄弟团队，方便他们构建自己的调用链分析系统。

◎进一步挖掘调用链历史数据的价值，力争在更多维度上提供出有价值的分析数据。



第5章
整体架构升级

5.1 ForceBot：全链路军演机器人
5.2 异地多活

京东是一家以技术为成长驱动的公司，自有技术平台在不断地变化、升级，促进了各类业务的增长。能否规划出真正解决业务问题的方案，取决于是否进行了合理的架构设计，并放眼未来，迎接变革——这就是架构的意义。

全链路压测，是这个时期应运而生的产物。京东的业务复杂，系统庞大，每年都要经受618和双11两次大考，而随着业务量及体积的逐年增加，每次大促对系统的能力要求也就不断提高。各核心系统环环相扣，其中任何一个出现问题，就可能会影响整个链路，直接影响用户购物体验，这也导致各团队在压测方面工作量非常大。此外，直接受影响的就是系统的容量规划。系统技术团队在2016年启动了全链路压测项目，覆盖所有信息系统，基于来自全国各地的真实流量，完全在线模拟用户行为，对全链路的各个系统进行高并发验证测试。此项目牵扯到京东研发体系的所有团队，跨团队协作、跨系统协调改造等工作量非常大，挑战性可想而知。

异地多活，不仅仅是建设一个机房，也是一个体系化的建设工程，影响着每个层级的业务状态，核心是“活”，通过更灵活的软件来定义数据中心，机房管理、基础网络、数据库管理、持久存储、中间件、应用部署、流量接入、监控与运维保障都有新的突破，各自体现着不同的价值主张，以多元化的互联解决个性化的需求。

本章所介绍的内容，并不是研发某一些系统，而是从全局的视角来推动京东商城整体架构的升级。

5.1　ForceBot：全链路军演机器人

5.1.1　ForceBot愿景

诞生背景

伴随着京东公司业务的不断扩张，研发体系的系统也随之增加，各核心系统环环相扣，尤其是强依赖系统，且上下游关系等紧密结合，其中一个系统出现瓶颈问题，会影响整个系统链路的处理性能，直接影响用户购物体验。往年的618、双11大促备战至少要提前3个月时间准备，投入大量的人力物力去做独立系统的线上压力评测，带来的问题就是各个性能压测团队工作量非常大，导致压测任务排期，压测的数据跟线上对比不够准确，各个强依赖系统上下游需要在压测中紧密配合，一不小心就会影响线上，有的在线下测试环境压测，压测出的数据更是跟线上差距太大，只能作为参考。更重要的一个问题是各系统的容量规划，每次大促前备战会必不可少的讨论话题就是服务器资源申请扩容的问题，各团队基本都是依据往年经验和线上资源使用率给出评估量，提出一个扩容量需求，导致各个业务系统每次促销的扩容量都非常大。为了解决以上各种苦恼，基础架构部在2016年整体牵头启动了ForceBot全链路压测项目，此项目牵扯到所有京东研发体系团队，各系统必须对压测过来的流量和线上正式流量进行区分标记等特殊处理，以识别两种不同的流量，不能因为压测流量而影响正常的用户体验和污染线上数据等，由于跨团队协作之多、跨系统协调改造等工作量非常大，挑战性可想而知！

能做什么

2016年主要实现了如图5-1所示的订单前的所有黄金链路流程高并发压测用户行为模拟，模拟用户操作涉及的模块包括首页、登录、搜索、列表、频道、产品详情、购物车、结算页、京东支付等，详细链路示意图如图5-2所示。在黄金链路中有各种用户行为场景，比如一般用户首先访问首页，在首页搜索想要的产品，翻页浏览，加入购物车、凑单、修改收货地址、选择自提等。各系统压测量以往年双11峰值作为基础量，在此基础上动态增加并发压力；同时，要区分对待两大压测的场景：日常流量场景和大促流量场景。大促场景下抢购活动集中，交易中心的写压力最大，且用户行为和日常有很大的反差，比如用户会提前加入购物车、选择满减凑单、集中下单等场景。

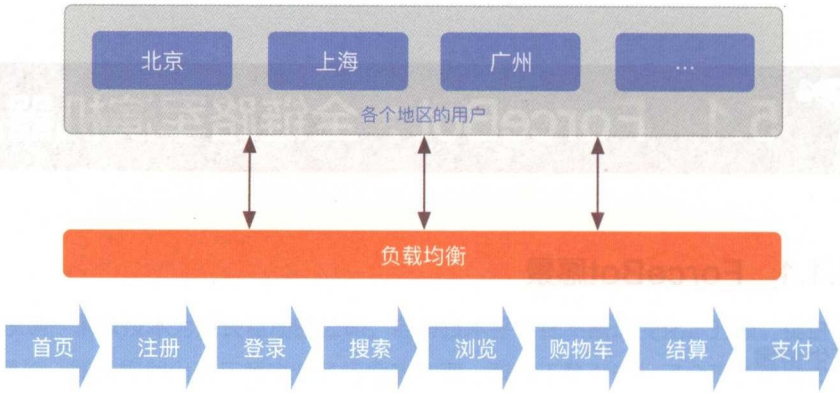


图5-1　黄金链路图

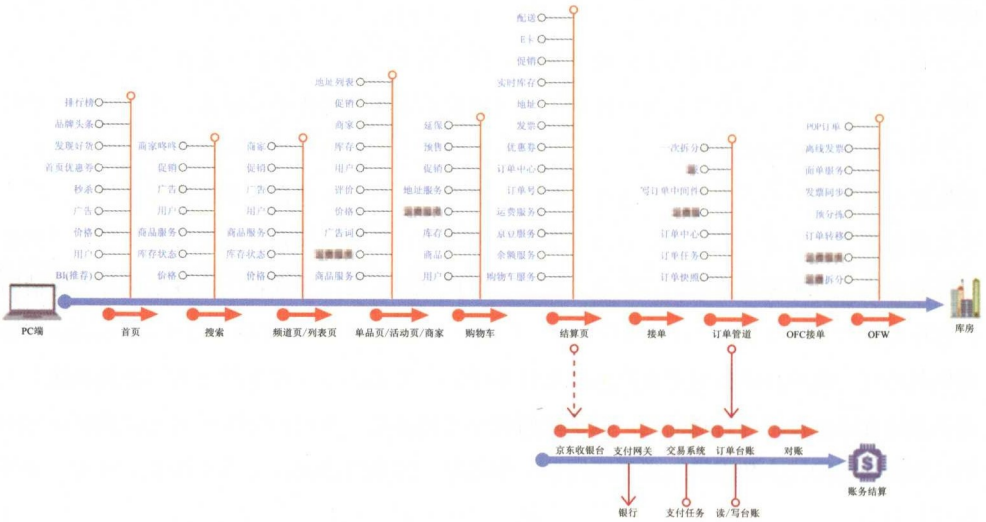


图5-2　京东全链路压测关系图

价值体现

ForceBot替代了往年各系统独自优化、性能压测备战状态，目前所有的备战数据和各系统性能承载能力、资源规划等都由ForceBot给出直接数据作为依据。以2017年618大促为例，5月份，我们实施了4次ForceBot军演压测，作为大促前性能优化与资源规划的主要依据。在军演压测过程中，秒级监控到压测源、压测中、京东所有的黄金链路系统、接口响应时间、TPS、TP99等数据，军演完成后提供了丰富的压测报告，准确地找到了各系统的并发瓶颈。

ForceBot同时也承担了内网单一系统的日常压测任务，开放给开发与测试团队，作为统一平台来支撑京东所有的性能压测需求，显著提升了效率。

5.1.2 ForceBot技术架构

工欲善其事，必先利其器。全链路压测必须要有一套功能强大的军演平台，来实现自动化、全链路、强压力的核心目标。

ForceBot架构

如图5-3所示，平台在设计时针对核心功能进行了尽可能的解耦，将各模块拆分成一个个相对独立的服务程序，所有服务均为分布式，避免了单点故障对系统功能带来的影响，并方便进行有针对性的水平扩容。其中一些服务程序说明如下。

◎Controller负责任务分配，并通过数据库字段维护Agent状态。

◎Task Service负责任务下发，支持横向扩展。

◎由Agent注册心跳、拉取任务、执行任务。

◎Monitor Service用于接受Agent上报的监控和日志数据并转发给Kafka，由Compute Service对数据进行进一步计算处理。

◎Compute Service对压测数据做计算，将计算结果保存到Elasticsearch。

◎Git用来保存性能测试脚本，提供完整的版本控制管理功能。

◎Script Package Service 为压力测试脚本提供统一的异步构建打包支持，通过对Maven的支持，与内网Maven私服交互，为脚本提供灵活可靠的依赖管理。

◎JSS为FoceBot提供了可靠高效的一站式资源分发解决方案，所有经Script Package Service构建并打包后的性能测试脚本会上传至JSS的对象存储，通过Agent主动拉取的方式向Agent分发资源。

◎系统为性能测试脚本提供了多种生命周期控制，已适用不同的场景，并可大幅度提升执行效率，减少对象创建次数。同时提供动态增减进程线程数量及Agent数量的功能，以灵活控制测试压力 。系统同时提供集合点设计功能，以适用更多场景。

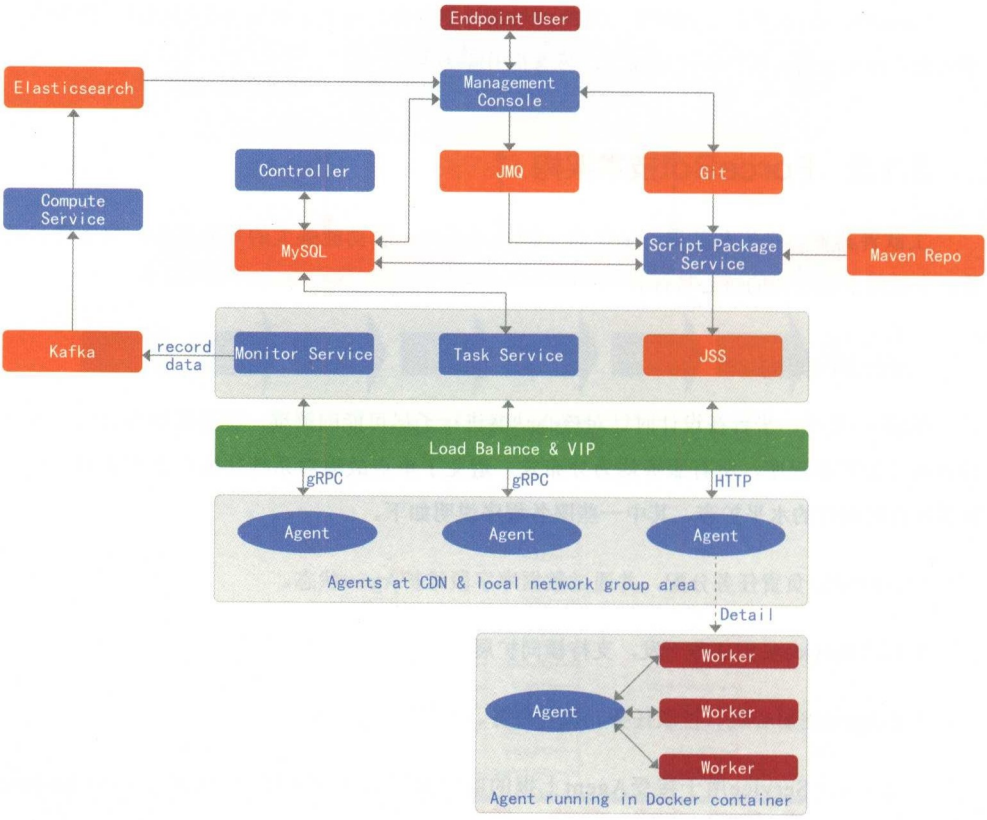


图5-3　ForceBot架构

核心功能

平台核心功能在于需要向性能测试人员提供一个高效的可操作环境，用于准确地描述其测试逻辑，并兼容大部门公司业务服务调用方式和场景。京东内部业务系统大部分使用Java语言开发，使用基础平台中间件技术部开发的JSF、JMQ等中间件进行服务调用，为此提供了一个与Java语言高度兼容的脚本语言执行环境作为性能测试逻辑编写基础，这一点尤为关键。

系统选用了兼容JSR223规范的Groovy作为主要脚本语言，并效仿Java下著名的单元测试框架Junit的设计哲学设计了一套高效友好的测试逻辑开发和执行环境。

平台核心脚本引擎为性能测试脚本设计了多种生命周期控制，以适用不同的场景，并使性能最优化。在脚本编写过程中，用户仅需要在Groovy脚本中使用内置的几种注解便可对脚本的执行和数据采集进行精确灵活的控制，如测试类生命周期、事物、执行权重等，大大提升了脚本开发效率。

容器部署

为了快速创建测试集群，Agent采用Docker容器的镜像方式进行自动化部署。这样做的好处如下。

◎利用镜像方式，弹性伸缩快捷。

◎利用Docker资源隔离，不影响CDN服务。

◎运行环境集成，不需要额外配置运行所需的类库。

◎每个Agent的资源标准化，能启动的虚拟用户数固定，应用不需要再做资源调度。

◎压力机可以以快速镜像的方式上下线，快速释放CDN资源。

心跳和任务下发

Task Service为Agent提供了任务交互和注册服务，主要包括Agent注册、获取任务、更新任务状态。

Agent启动后就会到Task Service注册信息，然后每隔几秒就会有心跳拉取一次任务信息。Controller会根据注册的信息和最近心跳的时间来判断Agent是否存活。

Agent在拉取任务和执行任务的过程中，会将任务状态汇报给Task Service。Controller会根据任务状态来判断任务是否已经结束。

Task Service采用了gRPC与Agent进行通信，通过接口描述语言生成接口服务。gRPC是基于HTTP2协议的，序列化使用的是protobuf3，并在标准单向RPC请求调用方式外，提供了双向流式调用，允许在其基础上进一步构建带状态的长连接调用，并允许被调用服务在会话周期内主动向调用者推送数据，其Java语言版采用Netty 4.0作为网络IO通信。使用gRPC作为服务框架，主要原因有如下两点。

◎服务调用有可能会跨网络，可以提供HTTP2协议。

◎服务端可以认证加密，在外网环境下，可以保证数据安全。

Agent实现

Agent采用多进程多线程的结构设计。主进程负责任务的接收、预处理和Worker进程的调度。将任务的控制和执行进行进程级别的分离，这样可以为测试的执行提供相对独立且高度灵活的类库环境，使不同任务之间的类库不会产生冲突，并有益于提升程序运行效率。

Agent与Task Service保持通信，向系统注册自身并获取指令。如图5-4所示，根据任务，需要启动Worker进程执行任务，主进程负责管理Worker进程的生命周期。Worker进程启动后会通过TCP连接与主进程保持通信，获取新的变更指令，如线程数变化通知，并及时进行调整。

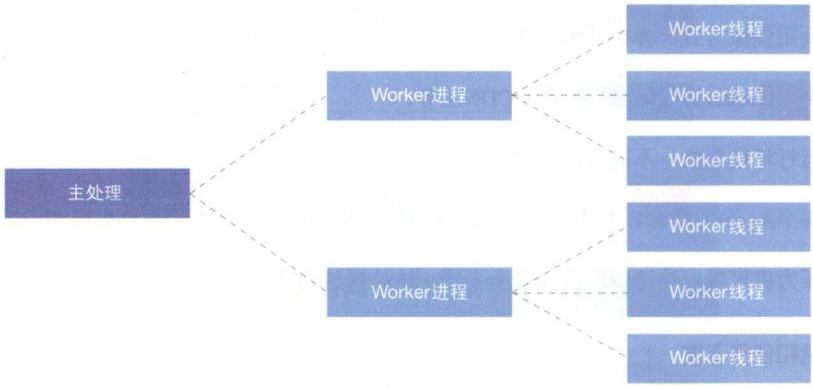


图5-4　ForceBot Agent结构

数据收集和计算

实现秒级监控。数据的收集工作由Monitor Service完成，也是采用gRPC作为服务框架。Agent每秒上报一次数据，包括性能、JVM等值。

Monitor Service将数据经Kafka发送给Compute Service，进行数据的流式计算后，产生TPS，包括TP999、TP99、TP90、TP50、MAX、MIN等指标，并将其存储到ES中进行查询展示。

为了计算整体的TPS，需要每个Agent把每次调用的性能数据上报，这样便会产生大量的数据，Agent对每秒的性能数据进行了必要的合并，组提交到监控服务以进行更有效的传输，如图5-5所示。

图5-5　数据压缩

5.1.3　业务系统改造

黄金流程业务

首期识别从用户浏览到下单成功的黄金流程，其包含的核心业务如下。牵扯的核心业务如图 所示，涵盖首页、登录、搜索、商品详情、列表页、购物车、支付等环节，上下游强依赖，少则几个接口，多则几十个。一个测试请求从最前端Web进来，如何让提供服务的所有系统都识别到这是测试请求？这将是一件重大的系统改造工程，基本涉及了全公司的核心系统。

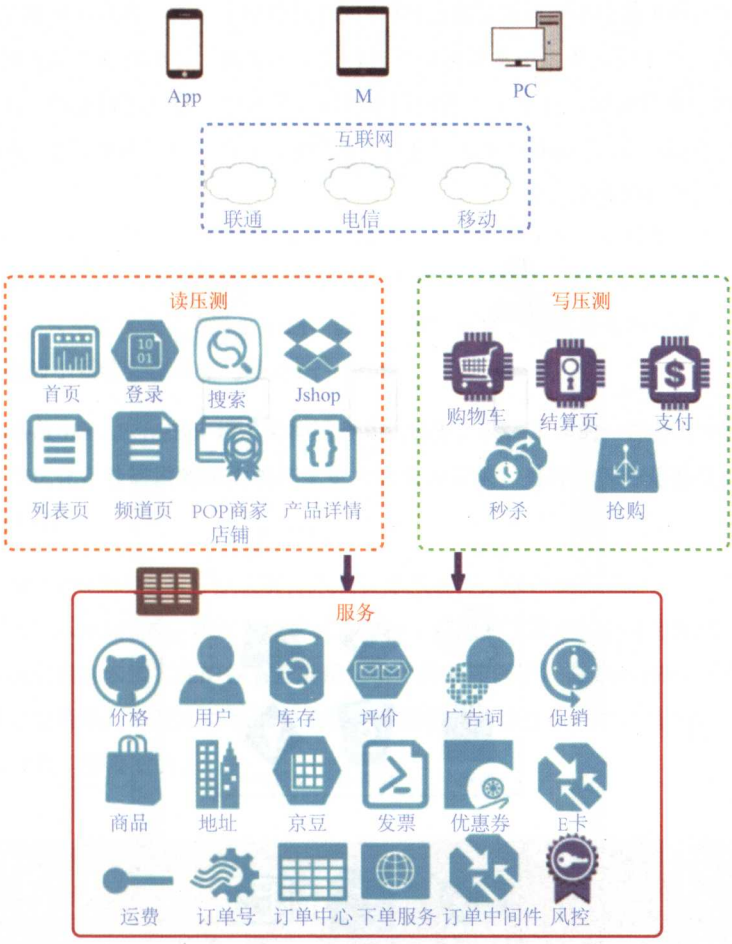


图5-6　压测关系

压测流量识别

压测流量是模拟真实用户行为从公网发起的流量，如图5-7所示，有GET请求、POST请

求。并发量都是历史大促峰值的数倍，要保障在军演过程中不能污染线上各种数据，比如PV/UV、订单量、财务报表、大数据分析等，更不能影响正常用户下单的购物体验。压测场景由两大部分组成：订单前和订单后。

订单前就是用户在确认购买商品提交订单以前的行为，比如打开首页、搜索、登录、浏览等称之为订单前，订单后是用户提交订单、修改收货地址、使用优惠券、填写发票信息、选择配送方式和支付方式等行为生成订单后的行为，订单后会有订单跟踪等系统信息更新，前者读多写少，后者写多读少。所以，对于订单前我们会在压测脚本开始入参标识，由最前端接收请求的Web识别到压测标识后通过JSF中间件透传给上下游，各自系统独立按正常逻辑处理请求，测试请求统计类系统拿到标识后进行过滤不计数。订单后各系统根据订单中间件对订单信息增加测试标识，订单生产系统识别测试订单后进行独立逻辑处理，正常生产。风控系统针对用户pin、测试sku和订单标识进行多重判断，以防正常订单被识别成测试订单或者测试订单标识丢失等问题的发生。

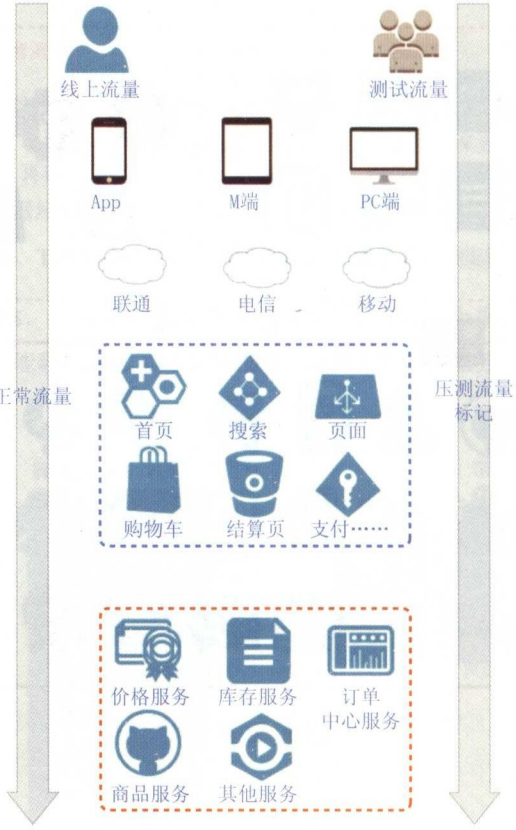


图5-7　流量隔离

压测数据存储

业务系统根据标识识别到压测流量需入库后，根据不同的场景，采用两种方式来存放压测数据。

◎打标数据并存储到生产库中，针对生产库增加字段标识压测数据，压测的数据能体现生产环境性能。压测后及时清理测试数据。统计报表要过滤掉测试数据。改方案能利用现有资源，需要系统进行较多改造。

◎构造压测库环境，根据压测流量，把压测数据存放到压测库中进行隔离。改方案需要较多的资源，但系统改造量小，可以根据压测流量路由到测试库。支付系统最大的改造困难就是银行接口的强依赖，不能用真实的银行卡扣款和支付，ForceBot的目标不是压银行接口，而是压自己本身的支付系统，所以京东的支付团队目前是自己构造了一个模拟环境，通过前端传递过来的压测标识，自动路由到模拟环境进行扣款支付。

全链路压测可以理解为网络链路 + 系统链路，网络链路是用户到机房的各个网络路由延迟环境，系统链路是各个系统之间的内部调用关系和强依赖性。 其实在去年双 11，京东就采用了 ForcBot，只不过仅仅针对重要链路，因为如果没有核心系统首先参与进来，小系统是做不起来的，因为小系统强依赖核心系统。

2017年备战618，ForceBot 技术架构没有太大调整。我们工作的一部分是平台用户体验的优化和性能的优化，让有限的压力机产生更大的压力请求；另外一部分是整合被压测的系统监控，实时展示。

京东希望 ForceBot 未来可以实现“人工智能预言”。现在还在逐步实现，我们希望未来的全链路压测引入 AI 技术，通过人工智能预言各个系统的流量值和资源分配建议，根据线上的系统军演数据预言未来大促的各系统场景。举个简单的例子，在 ForceBot 平台上录入每秒一千万并发订单场景，以现在的系统去承载，各系统是一个什么样的性能指标，以及瓶颈点在哪？ 这就是我们想要做的。

5.2　异地多活

5.2.1　概述

大型互联网公司的数据中心架构，通常都会经历从单机房到同城双机房，再到异地灾

备，最后到异地多活的过程。所谓“异地多活”，是指分多个地域、多个数据中心运行线上业务，并且每个IDC均提供在线服务。然而，不同公司因为业务特点不一，各自实施异地多活的目标与技术路线也不尽相同。

推动京东商城实施异地多活架构升级的目标与出发点如下。

◎首先，搭建商城业务的广域分布架构、实现数据中心级弹性扩展能力。即多地域多IDC，业务规模不受限于单个IDC容量，随IDC增加而弹性扩展。

◎其次，流量就近接入，灵活调度，进一步提升可用性与用户体验。

我们的技术路线如下。

◎首先，通过中间件系统的整体技术升级，做到对应用层完全透明。

◎其次，不显著增加成本，特别是包括数据库与商品图片的持久存储层机器成本。

5.2.2 Datacenter-Level Elasticity（数据中心级弹性扩展）

我们把异地多活架构升级，分成6个层面的技术工作：基础设施、持久存储、中间件、应用部署、流量接入、监控与运维保障（如图5-8所示）。按照从前台到后台的顺序依次实施：无线业务、搜索推荐广告、交易、订单后与物流。

图5-8 异地多活分层架构示意图

基础设施层

基础设施层分在华北、华南、华东三个区域（region），IDC数目分别为3、1、1。

IDC规模异构：各个IDC容量并不相等也不需要相等，大的机房包含上万台服务器，小的机房则部署千余台服务器。

机房之间专线互联，网络质量稳定。

持久存储层

持久存储层主要包括MySQL数据库和商品图片。

可靠性目标：任何1条记录，至少存储3份，至少跨2个IDC。主IDC内同步复制，异地IDC异步复制。

成本目标：×3，而不是×*N*（*N*为机房数目），即无论机房是5个还是6个，投入的数据存储最多是3份。

弹性数据库JED本身就是为异地多活而设计的，因此内置支持多地域多IDC的部署与管理，对遗留的并非接入JED的MySQL实例，进行了拓扑改造。商品图片底层存储JFS，在2015年到2016年进行了架构升级，从之前的单IDC三个副本同步复制，升级为支持异地多机房，其复制协议也如图5-9所示。

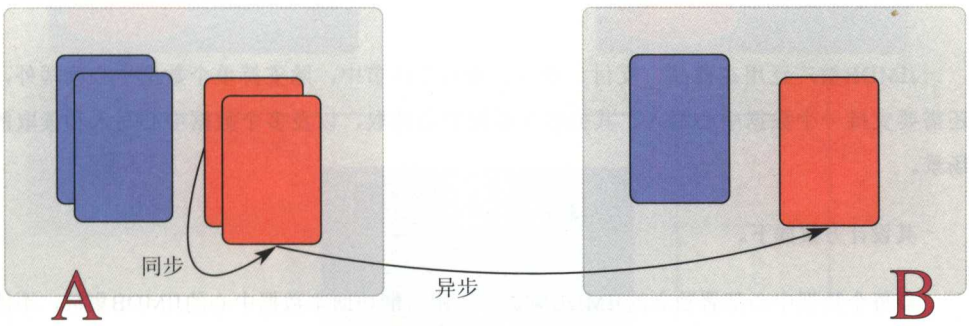


图5-9　数据复制范式

中间件层

中间件层主要包括JIMDB、JMQ、JSF。

改造目标：JIMDB实现一处写入、全局同步，IDC之间低延迟多向复制，且JMQ和JSF支持异地多机房部署。

应用层

应用层的具体应用代码不需要改造，而需要设计应用域划分——哪些应用部署在哪些region、哪些应用具有亲缘性。

流量接入和调度

流量接入和调度主要包括 GSLB、HTTPDNS等。需要能够根据用户网络质量，动态调度用户就近访问。

运维保障

运维保障对各个IDC及IDC之间的专线进行实时监控，确保发生故障时可以触发failover与流量调度。

5.2.3 中间件系统改造

之前，JIMDB与JMQ等有数据状态的中间件产品在设计时并未考虑异地多活，因此需要进行改造升级。

JIMDB

JIMDB被广泛用在商品、支付、推荐、搜索等环节中，除支持单个数据中心读写外，还需要支持一个数据中心写入、其他多个数据中心读取，以及多个数据中心写入和读取的场景。

其设计方案如下。

◎每个数据中心部署独立的JIMDB集群，应用只能访问本数据中心的JIMDB集群，不能跨机房进行访问。

◎当有数据写入时，JIMDB服务端会记录复制日志，外部的复制程序拉取复制日志，把数据转发到另一个数据中心的集群中。管理端会记录好需要同步数据的集群关系，有复制关系的两个集群之间，数据相互进行复制。数据中心D1与数据中心D2分别部署了集群C1和C2，C1和C2之间有复制关系，当数据a由业务应用写到C1集群中时，数据a会通过复制程序被复制到C2集群中，由于C2的数据也会复制到C1中，因此为了防止数据a被复制回C1，数据a在写入时会标记写入的集群标识，不是通过业务应用写入的数据，不会被复制到其他集群中，这样就防止了数据a在集群间被反复

复制。

◎JIMDB数据保存在内存中，异步进行持久化，考虑到写入性能复制日志的记录也会保存在内存中，但是内存空间有限，当复制程序由于网络、硬件故障等原因发生中断后，有可能导致内存空间不够，会淘汰掉还没有复制的日志，而当故障恢复后，由于复制日志有丢失，因此只能进行全量数据复制，无法进行增量复制。数据中心之间一般距离较远，发生网络故障的概率会比较大，另外全量同步数据也会比较慢，因此当出现日常故障时尽量不要发生全量同步，减少数据中心之间的带宽压力，也缩短故障恢复的时间，复制日志在复制发生延迟时需要持久化。

◎由于集群之间除了复制程序外，相互之间不需要进行交互，因此集群的支撑模块可以和服务端一起部署在同一个数据中心，系统架构图如图5-10所示。

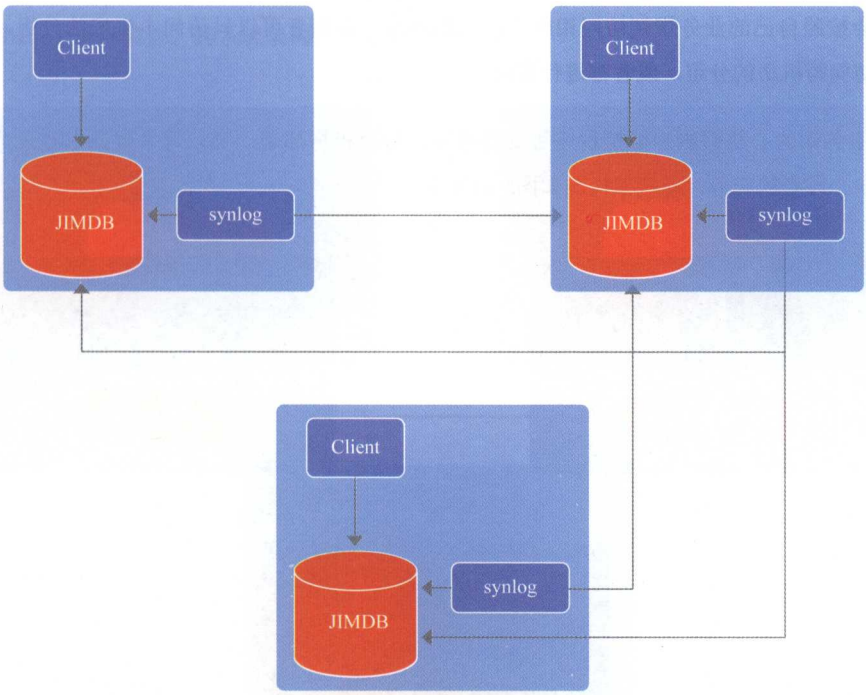


图5-10　JIMDB异地多活架构图

JSF

JSF作为系统间相互调用的RPC框架，需要提供跨数据中心访问的能力。服务的提供方和服务的调用方可能部署在同一间机房，也可能部署在不同的机房。因此，服务的注

册信息需要广播给所有的数据中心，当有新服务发布时，会先在本数据中心的注册中心进行注册。其他数据中心的注册中心接收到有新服务发布的通知后，也会在本数据中心记录该服务发布的数据中心、IP信息、接口名等信息，因此每个数据中心的注册中心会有全量的服务信息。当然，如果有服务只希望在本机房内被访问，则注册信息可以不进行广播。

服务的注册信息在各个注册中心进行同步时，需要保证通知的时序性，比如一个服务发布后，由于某种原因下线了，如果注册中心先拿到下线的通知，后获取到发布的通知，就会导致状态不一致。对时序的保证，可以通过版本进行控制，低版本的通知不能覆盖高版本的通知，为了保证通知的信息不会发生部分更新，单个服务的注册信息需要全量告知给其他的注册中心，进行整体替换。

服务端在注册时会按照数据中心和业务规则进行分组，调用方从注册中心拿到注册信息后，会按照自己的业务规则访问相应分组的服务端。调用者可以只访问本机房的分组，也可以只访问跨机房的分组，或者都进行访问。

服务端的心跳检测和监控程序等支持模块，每个机房部署一套，每个数据中心只负责本数据中心发布的服务，系统架构图如图5-11所示。

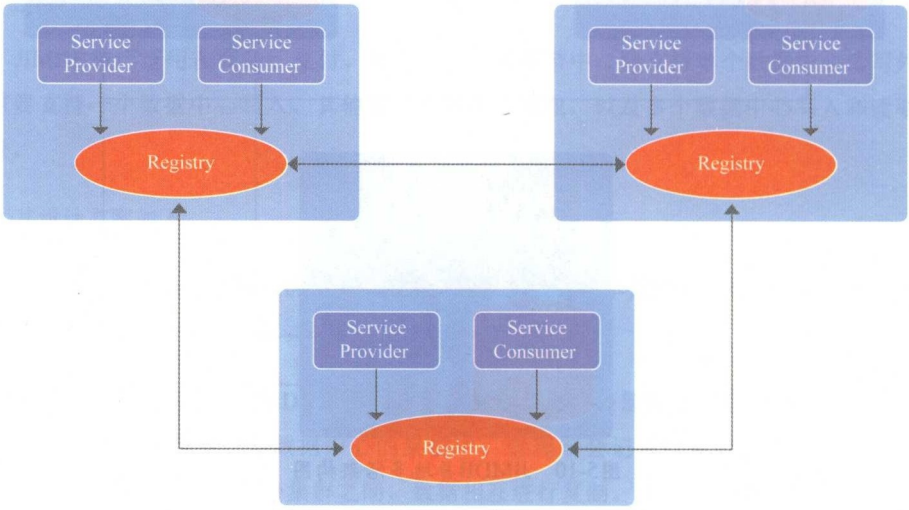


图5-11　JSF异地多活架构图

JMQ

JMQ作为消息中间件，消息只是暂存在Broker端，消息消费完成以后就不会再被获取，因此消息被取走后，如果没有对消息进行重放的需要，就可以马上将其移除。因此在大部分场景下，消息不需要在数据中心之间进行复制。生产者和消费者与Broker部署在同一个数据中心。

对于极个别十分重要的消息，为了防止消息积压时发生数据中心整体故障，可以在另一个数据中心部署一个远程的Broker进行数据备份，当故障发生后可以切换备用的消费者消费这个备份在Broker上的消息。

当业务发布一个消息主题时，选择主题需要分布的数据中心和部署在这些数据中心上的Broker，Broker之间没有关联关系，相互独立。生产者只会发送消息到本数据中的Broker上，消费者也规定只能从本数据中心的Broker上获取消息。当生产者和消费者不能部署在同一个数据中心时，则需要跨数据中心部署Broker Group，一个Broker Group是Master-Slave关系，消费者从Slave上进行消费。其系统架构图如图5-12所示。

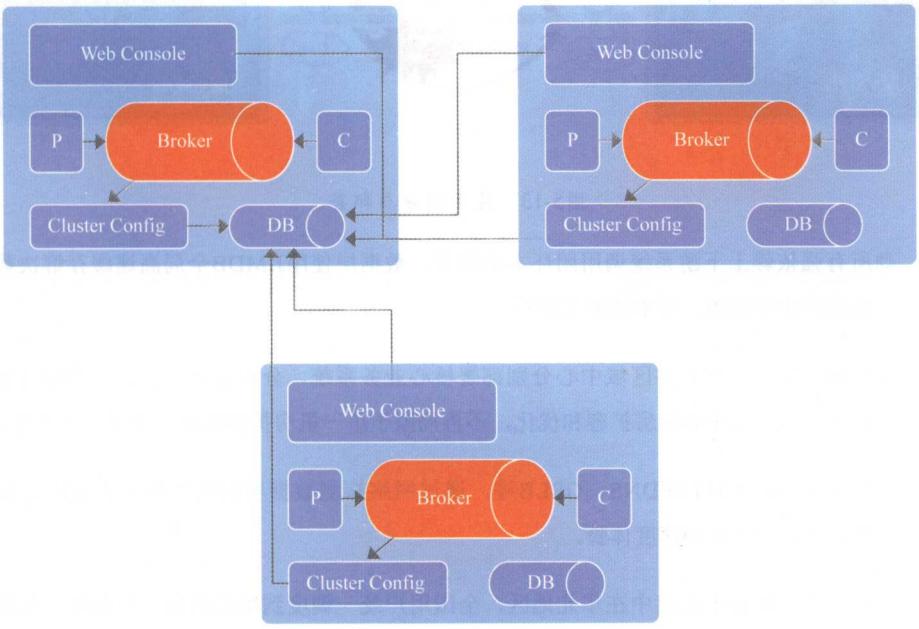


图5-12 JMQ异地多活架构图

5.2.4 应用划分与用户就近接入

目前，京东应用基本都部署在了Docker环境中，以镜像的方式发布和更新系统，完全脱离了物理机时代的上线发布系统，更是集成了负载均衡、DNS、日志、监控等重要工具，为研发和运维提高了工作效率，让各中心的业务部署变得更加简单。

应用层部署按如图5-13所示开展。

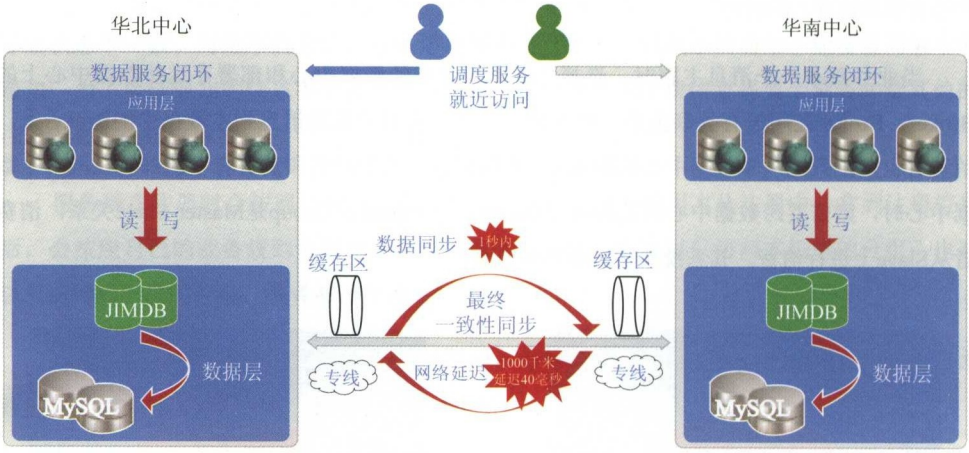


图5-13 应用划分与部署

◎所有强依赖上下游系统调用闭环本地部署，数据层使用JIMDB全局高速缓存解决数据延迟同步等问题，并本地闭环读写。

◎华南、华东、华北各区域中心分别部署核心业务系统，分担总体流量，日常情况下分别在三大区域中心机房扩容和优化，不再局限于单一机房扩容机柜、带宽、电力等。

◎调度服务，如HTTPDNS、GSLB等，通过网络探测数据实时调度和优化就近访问策略，保障用户访问速度体验。

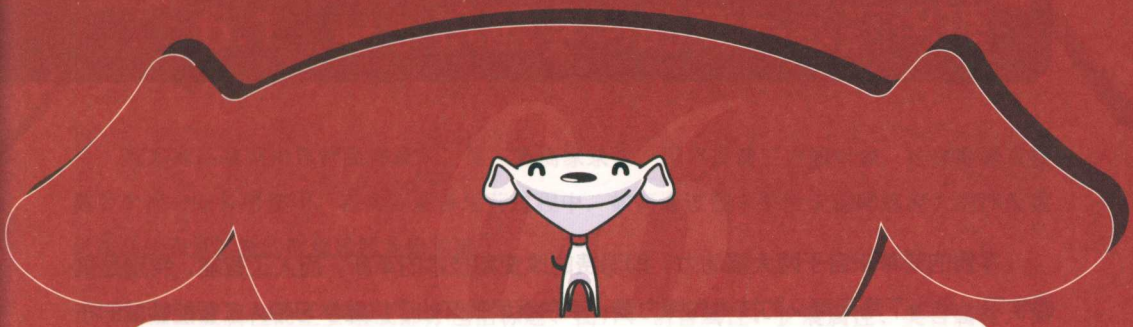
以前的京东数据中心集中在华北地区，全国用户统一调度到华北机房进行服务，如遇到南北互通、运营商专线异常等网络故障，只能通过切换用户入口跨运营商进行优化。中国的互联网比较复杂，南北互通问题是多年的痛点。而异地多活，则能使用户流量就近接入。关键点说明如下。

◎京东目前在国内有华北、华南、华东三大区域，每个区域为一个中心，三大中心服务于国内用户，每个中心根据规模和覆盖规模不同，可以以本地区IDC级别扩展多个机房。

◎每个中心有联通、电信、移动和BGP带宽接入本中心，依据GSLB、HTTPDNS和全国网络质量监控数据，动态优化用户到某个具体中心提供服务。

◎中心之间用长途专线打通，分别互联其他两大中心，通过长途专线数据秒级同步，提供低延迟数据最终一致性保障。

◎每个中心之间的专线距离大约1000千米左右，专线延迟在40毫秒左右。在此约束下，JIMDB实现数据快速同步，是异地多活整体技术堆栈的核心部分。



第6章
机器学习技术

6.1 基于机器学习的商品数据治理

6.2 智能分单

6.3 列表页排序

6.4 语音识别与客服导航

6.5 商品上新助手

零售的变革来自于两大驱动力，即消费的改变和技术的革新，而人工智能，特别是机器学习，是当下毫无疑义的技术先锋力量。京东很早就开始了机器学习和人工智能领域的布局，在2014年9月，京东便组建了深度神经网络实验室（DNN Lab），后来成为基础架构部机器学习技术团队的前身。

从基础架构部的角度来看，随着GPU等硬件资源的革新，以及深度学习算法的成熟，机器学习技术已经不再是实验室的一项研究，在工业领域也有非常广泛的应用前景。

从业务角度来看，京东的业务一直在不断高速增长，各个环节继续增加人力显得无以为继，也对运营成本提出了巨大的挑战。而且在人力增长的同时，知识和经验又无法在人与人之间快速精准地大量复制，从而导致运营的不稳定性对用户体验造成伤害。因此，我们迫切地需要用“数据”和“算法”来取代一部分的“经验”和“人工”，提高运营效率，改善用户体验。本章将会围绕成本、效率、体验，从应用的角度，带给你机器学习技术在零售行业的最佳实践。

机器学习技术的应用，提升了电商各环节的效率。而应用的需求和场景的不断扩展，又反过来促使了技术的进步。通过实践和迭代优化沉淀出来的计算机视觉、自然语言理解和语音能力，不仅可以服务京东内部各个部门，还可以加工形成人工智能的通用技术和服务，对外输出。本章会介绍京东在这方面的实践。

6.1　基于机器学习的商品数据治理

京东商品数据由自营品牌录入、第三方商家录入、用户反馈三方面组成。这些数据直接被多个核心业务所使用。在实际业务运营过程中，如何通过技术手段来保障这庞大的涉及数亿商品的海量数据的质量显得尤为关键。

商品工业属性的主要组成部分包括标题、图片、销售属性、扩展属性、类目属性，如图6-1所示。但是由于各种各样的因素，例如抢占搜索命中的概率、商品上架的随意性等，商品数据存在商品类目错误绑定、图片与文本属性不一致、商品标题短语堆砌、上传图片违规等各类问题。这些问题共同组成了目前业务部门的痛点。

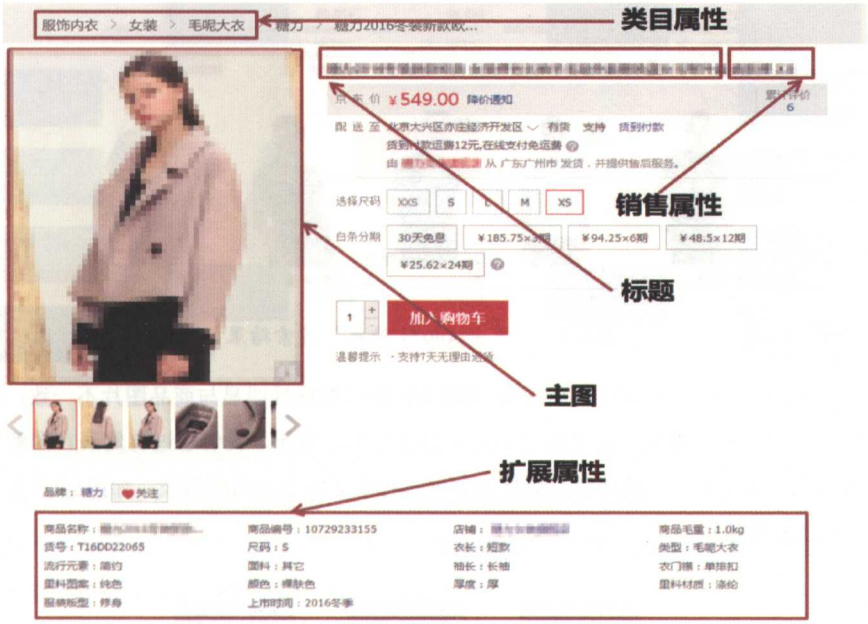


图6-1　商品工业属性

我们的机器学习应用就起步于解决这一类痛点问题。下面将分别用具体的案例来简要介绍在解决这些不同的问题时所涉及的思路和技术。

6.1.1 商品图文不一致校验

图文不一致性是指商品图片与商品文字描述不一致的情形，通常是商品图片和描述商品属性的一个或者多个文本存在冲突。商家在录入商品时，除了上传图片外，还需要添加一些文字信息对商品进行说明，比如商品标题、商品的属性信息等。有些商家由于操作失误或出于SEO的目的，在录入商品时添加了虚假或者扩大宣传的信息，在商品的原始数据里引入了噪声，基于这样的信息做索引和搜索便会影响搜索的精度，进而影响用户的体验。

举一个简单的例子，当我们在京东主站搜索 “男士polo衫 纯色”时，出现了一些非纯色的polo衫，如图6-2所示。

图6-2 “男士polo衫 纯色”搜索结果

图6-2中最右下角的格纹衬衫的扩展属性为纯色。该属性明显与商品图片不一致，错误的数据降低了搜索的准确率，但是增加了商品的曝光率。为此，我们通过自然语言理解和图片识别技术来检测图文不一致现象，对错误数据进行修正并通知商家修改。如图6-3所示，为图文不一致的检测流程。

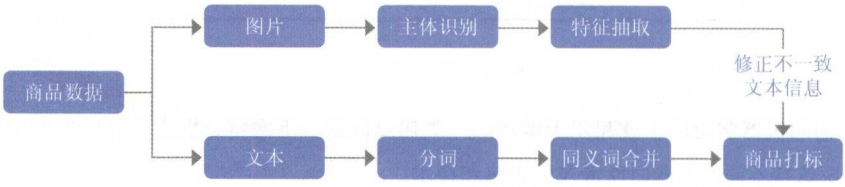


图6-3 图文不一致检测流程

其中的关键就是图片识别技术，从商品图片中识别商品主体并抽取商品的特征，为文本属性修正提供高置信度的参照。以服饰为例，从主图上就可以识别颜色、袖长、裙长、图案等属性。

商品图片的属性识别依赖高质量的训练数据。商家在描述商品颜色属性时五花八门，形容红色的就有“绯红”“桃红”“品红”“鲜红”等，品类繁多，总的属性值达到上万个，而且很多颜色的区分度很小，即使请专业人士来看，肉眼也很难区分，这就给训练数据的收集带来了很大的困扰。因此，我们需要进行属性的归一化，如图6-4所示。这块主要依赖经验，对各个属性进行规整，如颜色我们分为了20个不同的色系，然后再分别进行数据的收集。

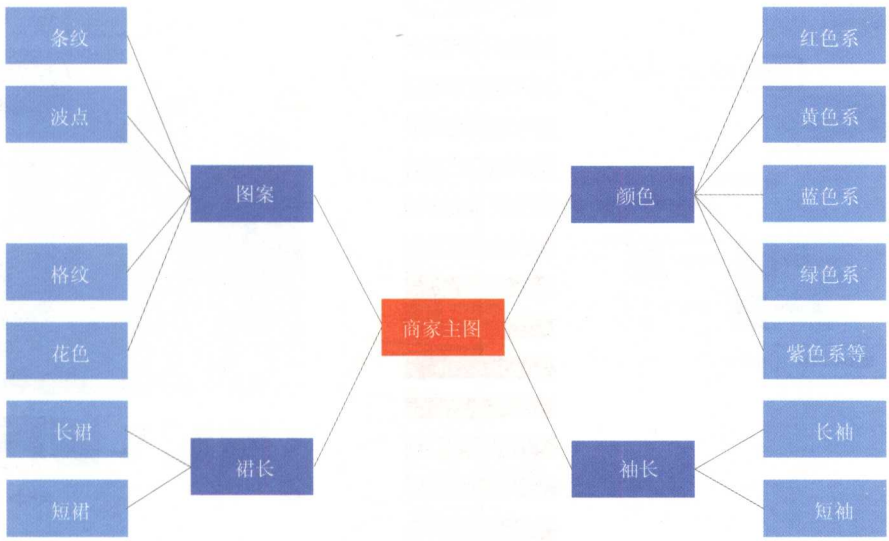


图6-4　属性归一化

接下来的事情就转变为图片分类问题了。图像分类一直是机器学习中比较热门的领域，在卷积神经网络提出后更是发展迅猛，目前图像分类的准确率在某些方面甚至超过了人类。在商品属性识别方面，我们使用的是业内较为领先的残差网络（Resnet）技术，如图6-5所示，在实现上针对自身图片特点进行一定的优化。

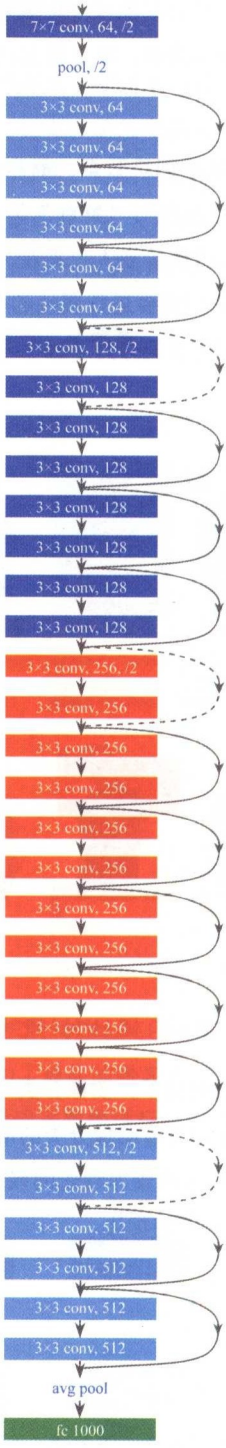


图6-5　商品属性识别网络模型

经过反复进行“训练—预测—不一致数据人工校验—错误数据加入训练集—训练”，不断迭代，最终使得在属性识别上的准确率超过了95%。基于此，我们对10%的商品进行了属性纠正，大大提高了商品数据的质量，提升了用户体验。

6.1.2　商品类目自动识别

每个电商网站都有自己的商品类目定义（如图6-6所示），同时，部分类目在发展过程中的拆分合并使得商品存在类目错绑问题。目前，京东有近4000个商品三级工业分类，这对于商家上架商品选择类目来说也是一个难题。

图6-6　商品类目

由于搜索、推荐、列表页等核心业务线都调用了商品的类目属性数据，所以为了降低类目错绑对核心业务的影响，优化商家上架商品的体验，我们进行了商品类目自动识别工作，模型可根据商家录入的标题，自动推荐所属分类，不仅简化了商家的上新流程，也减轻了类目监管的压力，使得运营越来越智能化。

在技术实现上，对word2vec的CBOW模型进行了创新性改造，构建了BTC模型，并加入了Dropout层，改造后的模型有效避免了训练过程中的过拟合问题，训练精确度明显提升。模型架构如图6-7所示。

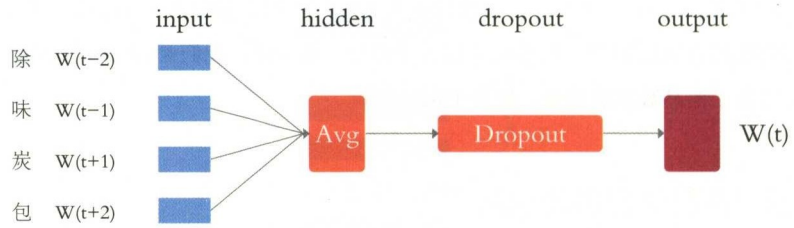


图6-7 类目自动识别架构

在多次迭代之后，我们获得了99%准确率的预测模型，应用方式如图6-8所示，比如提交“标题：××家纺单品被套天鹅绒保暖被罩”，返回“13818 被套”。

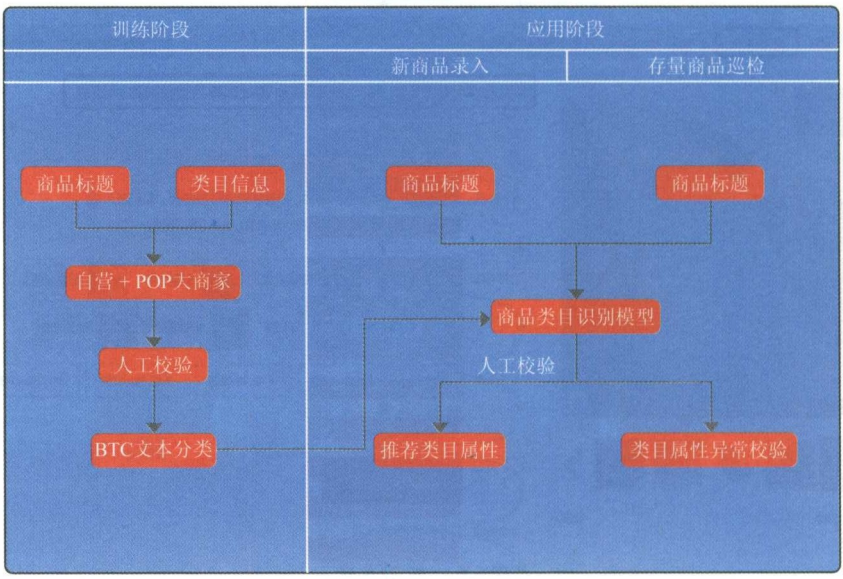


图6-8 类目自动识别应用

6.1.3 电商短文本理解：商品标题优化

机器学习中的自然语言处理技术不仅能够用来解决商品类目的自动识别问题，而且可以在深入了解商品标题的基础上，对商品的标题和属性描述进行优化。

商品的标题由采销人员和第三方商家命名，其中自营商品的命名较为规范。而第三方商品的标题由于没有一个严格的命名要求，导致标题除了有文本描述矛盾，还出现了大量词汇堆砌的情况，如图6-9所示。

运动鞋男士跑步鞋春季飞线休闲板鞋韩版透气男鞋子网布潮鞋潮流学生跑鞋2017夏季新款灰色（革面）40

图6-9　标题文本堆砌

从图6-9可以看到，一个跑步鞋商品的标题上出现了6次“鞋”。对于商家来说，如此命名是为了增加搜索、推荐等系统命中率。然而对于我们来说，需要对此类词汇堆砌严重的标题进行降权，使其出现在搜索结果的末端，避免影响到用户体验。这就需要利用机器学习技术对商品标题进行一系列的处理：分词、实体识别、属性打标、热度计算、中心判定，将商品标题结构化。在商家录入标题的过程中自动利用模型进行打分和结构优化，从源头上优化商品标题质量。

类似的应用场景还可以通过对商品标题的理解来提取出标题中所描述的商品属性，提取出的商品属性可以用来校验或者扩展商品的主属性数据，从而最终使用户得到更好的搜索和推荐体验。比如，商品标题“××短袖衬衫男2017夏季新品时尚格纹水洗浅绿格纹(TE)175/96A(41)”，通过我们的“电商短文本理解”模型可以识别出该商品所属分类、品牌、颜色、材质等信息，如图6-10所示。

图6-10　电商短文本识别模型

6.1.4　用户评论信息抽取

用户的交互，比如对于商品的反馈，是京东商品数据的一个重要补充。用户的反馈主要包

含商品评论、问答、退换货等信息。京东的理念是客户为先，因此我们对于用户的反馈极为重视，这些信息可以直观地展现用户对于商品的感受和质量的反馈。我们为此构建了多类语义理解模型，利用用户反馈实现选品、商品质控等。下面我们抽取一些评论信息进行着重介绍。

之前，用户在购买商品时，用户的评论对用户的购买意愿有着极强的影响力。但是在用户海量的评论中存在大量的无意义垃圾评论，这些垃圾评论对用户选购商品没有任何帮助，并且会影响到用户获取有效的评论信息。为此，我们通过构建语言模型对评论与商品的关联程度进行打分，将没有实质意义的评论折叠在评论最后。

此外，我们会从评论中抽取关键词短语，通过对短语与短语之间相似度的计算，聚合出频次最高的短语及相应的评论内容，展现给用户。更进一步，我们还会分析用户评论的情感，真实展现负向情感关键词，使用户获得一个直观的产品描述，诚信对待消费者。如图6-11所示，为评论信息抽取的框架。

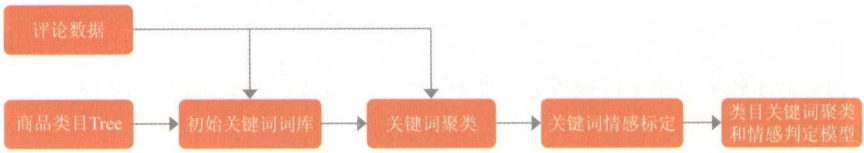


图6-11　评论信息抽取框架

技术实现上采用了无监督与有监督结合的方式构建关键词抽取模型。在模型设计中，首先考虑到单独的词汇表达能力有限，不能满足业务需求，比如“声音”“颜色”“外观设计”等，而我们更需要的是“声音大”“颜色艳丽”“外观设计时尚大方”等，所以采用了短语向量训练。然后根据业务的其他需求加入了语言模型、情感模型、功能词模型等，最终取得了不错的成果。模型还实现了定期自动更新，会自动对用户提交的评论做以上相关分析与处理，返回对应Top关键词。结果示例如图6-12所示。

```
1 画面比较清晰 140
2 看起来很大气 140
3 外形美观大方 108
4 质优价廉物美 102
5 大品牌值得依赖 72
6 看上去很漂亮 62
7 功能齐全 38
8 价格比较实惠 35
9 立体感很强 31
10 相信海信品牌 31
11 性价比比较高 29
12 做工精细 28
```

图6-12　评论信息抽取示例

6.1.5　商品详情页知识抽取

除了商品标题，在商品详情页中还包含商家对商品更多的描述信息，但京东的商品详情页以图片为主，这给文本信息的提取带来了很大的难度。对商品详情页图片进行OCR识别，从丰富的商品描述中提取其中的核心关键字，有助于构建完善的商品知识图谱，提升搜索质量。

不同于传统的扫描文档OCR，京东的商品详情页图片OCR存在如下挑战。

◎分辨率低：传统的扫描文档图像大多在200DPI以上，而网络商品图片为节省流量，一般都在72DPI左右，不太清晰。

◎压缩失真：图片大都经过了图像压缩，以节省空间。常见的格式是JPEG格式，由于丢弃了高频信息，所以会造成文本周围变形失真。

◎版面复杂：多字体、字号、颜色，文本行不一定水平，图文混排，版式随意，且可能存在复杂背景干扰。

◎中英文混杂：这是一个大类别集的字符识别问题。

上述挑战的存在，使得传统的OCR识别技术都失效了，因此我们采用了基于深度学习的商品详情图片OCR，由如下两部分构成。

1. 基于全卷积网络的文本行定位。由于商品详情图片的版面复杂，风格差异较大，传统的基于启发式规则的版面分析方法是行不通的，因此我们采用了在图像语义分割领域十分有效的深度学习模型——全卷积网络（Fully Convolutional Networks），来实现端到端的可学习的图片版面分析和文字定位算法，达到区分文字和背景区域的目标，如图6-13所示。

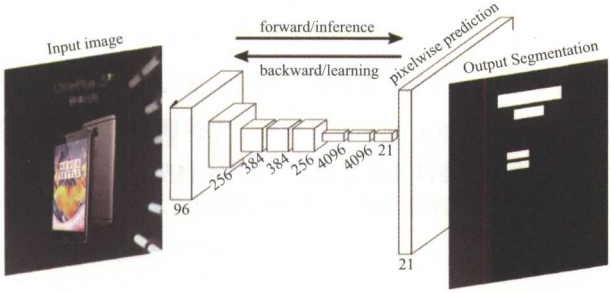


图6-13　全卷积网络的文本行定位算法架构

2. 端到端的通用字符串识别系统。如图6-14所示，通过CNN model获得图片的特征，与基于大规模语料数据训练递归神经网络（LSTM）的通用语言模型相结合，再通过基于时序的分类（CTC）输出。

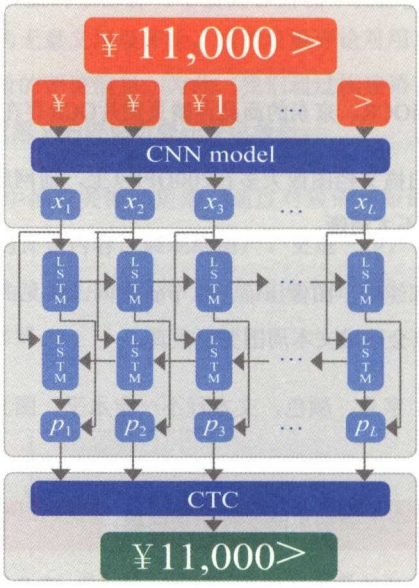


图6-14 端到端的字符串识别架构

端到端的文本检测与识别算法克服了传统OCR鲁棒性不足的问题，即使对于京东网站上各种压缩失真和版面复杂的图片，也能得到很好的文字识别结果。

6.1.6 商品质控

随着京东的发展，第三方商家的比重越来越大，第三方商家在丰富京东产品的同时，也给京东商品质量的把控带来了挑战。从最开始，正品行货就是京东的行为标准。对于自营商品来说，我们可以很容易从源头上控制；但对于第三方商家来说，如何从海量商品中精确拦截劣质商品，维护好京东生态并保护好消费者利益，成为一大难题。而其中最核心的问题就是我们能否通过用户反馈的信息，包括商品评论、退换修申请、客户投诉，找出质量问题大、疑似假货、二手货、水货的商品及高危商家。如图6-15所示。

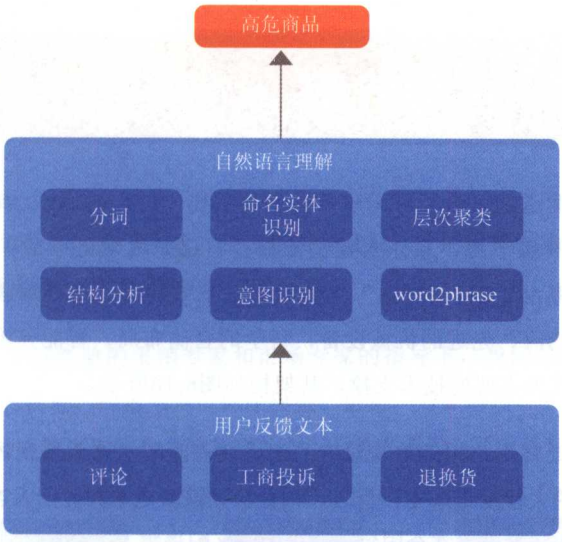


图6-15　商品质控

这其中涉及如下两个问题。

1. 通过评论、客户投诉文本等判断这件商品是否有质量问题，这是一个典型的文本分类问题。文本分类也是机器学习的核心问题，早期有SVM及其变种，但我们应用在这个场景的时候效果并不是很理想；而深度神经网络又出现了训练耗时过长等现象。基于此，我们采取了快速文本分类算法，它结合了自然语言处理和机器学习的一些先进理念，使用bag of words和n-gram表征语句，还使用了subword信息，通过隐藏特征在类别中共享信息，并通过softmax加速运算过程。最终，在质量问题检测上，算法实现了92%的准确率，而在假货、二手货、水货的检测准确率上更是达到了95%。

2. 对于如何从数以千计的评论或者客户投诉文本中找出负面关键字，帮助审核人员快速定位问题，我们采取了有监督和无监督相结合的方式提取关键短语，然后进行层次聚类，抽取核心负面关键字，将人工审核时间由原先的几分钟缩短到几十秒，大大提高了人工效率。

6.1.7　图片合规审查

图片合规审查主要包括诈骗二维码检测和黄色图片检测，此处不赘述。

6.2 智能分单

在电商行业，配送的时效性对用户体验极其重要。近年的618、双11大促期间，京东每天产生的订单就有数千万，京东自建物流需要自己完成配送环节。目前，京东在全国一线城市支持自营商品“211即时达”“上午买，下午到”。这种京东速度是如何达成的呢？它离不开自建物流、先进的机械设备、科学的仓储布局、智能分单、京东精神等的共同促进。智能分单作为主要的技术支撑，其架构如图6-16所示。

配送之前，需要将用户的地址与对应的配送地点匹配，这关系到物流效率最基本的问题。匹配率高会节约大幅配送成本，反之，则会极大浪费成本和时间且客户体验极差。基于传统匹配和gis的预分拣系统，在全国每天产生的跨区配送需人工分拣的订单约5万单，不仅耗费了大量的人力，还影响了配送的时效。通常，每个省份的站点都在数千个以上，对应着上亿条地址数据。我们基于每个省份的地址与配送站的关系建立了一个分类模型。最初，由于京东配送站的密集会造成相邻配送站在匹配时出现误匹配，同时，分类模型还有一些过拟合的问题。

通过后期优化，智能分单增加了分类神经网络的层数，同时限制一些不必要的学习机制以降低过拟合现象。在实际生产测试中，我们发现模型的分类准确率和鲁棒性都很好，对于地址重名、地址缺失、拼写错误都可以正确匹配，各省份模型在测试集中匹配时的准确率均值为99.43%。最终上线后，日均节省了北京地区93%的人工分拣单。

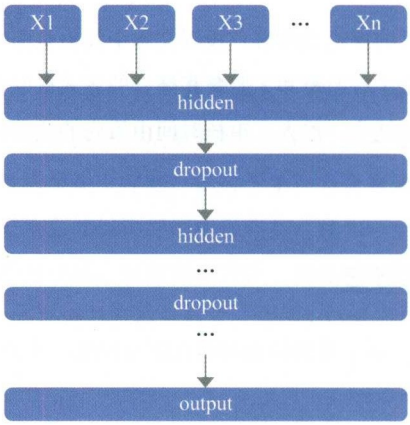


图6-16 智能分单架构

6.3 列表页排序

列表页是电商商品按照工业分类的展示页面，每天有海量的用户通过列表页浏览和购买商品，因此如何制定商品的排序策略，将流量合理有效地分配给高转化的商品，让用户可以更快速地找到心仪的商品是列表页排序的核心问题。

之前，在具备丰富经验的采销专家和市场专家的指导下，我们选取一部分商品相关维度，人工赋予一定的权重，构建列表页排序模型。但是，专家的指导往往导致一些隐藏的特征无法体现，同时，主观因素也会导致权重的不科学性和模型过于个性化，如图6-17所示。

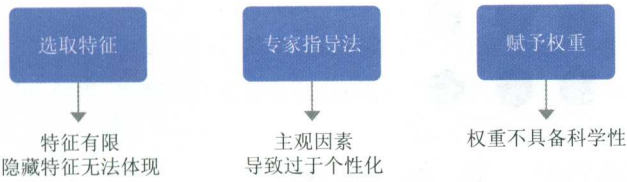


图6-17 专家指导法不足

通俗来说，专家对于模型的定义是通过记忆保留的丰富经验，和学习到的市场规则形成的。因此我们在探索机器学习智能化应用的过程中，要考虑可否通过联合训练一个具备一定宽广度的线性模型用于记忆，训练一个深度神经网络负责学习，并用于推广。通过集合二者的能力，我们使用Wide & Deep Learing模型来实现更为高效的集中推荐。它在做大规模回归和分类问题时可以选取大量的不同特征值，帮助我们发现更多的隐藏特征，实现更优的排序方式。

在列表页中，用户选择特定的类目，前端页面展示商品的排序。评价列表页排序好坏的重点指标是用户是否选择商品并且购买，且商品的排序是否尽可能靠前。

利用广度模型，可以训练一个线性模型，通过宽广的交叉乘积特征变换来记忆用户的喜好表现，学习数据集中在浏览列表页后下单且下单商品出现在列表页中的客户（正样本）的特征值。这个模型可以计算每一个商品被用户选择购买的概率P（选择购买），然后利用商品购买的概率来实现商品的排序。换句话说，广度模型在记忆用户的喜好上面表现不错。广度模型如图6-18所示。

但同时，用户对于一成不变的推荐商品容易厌倦，一些与众不同的商品可能会达到意想不到的效果。在实际测试中，我们发现选择三门冰箱的用户并不介意去浏览了解并最终选择一个对开

门冰箱。这时，训练一个深度前端神经网络模型为每个商品和类目找到一个低维稠密表示，来实现在嵌入空间对请求内容进行近邻匹配，可以优化列表页的排序。深度模型如图6-19所示。

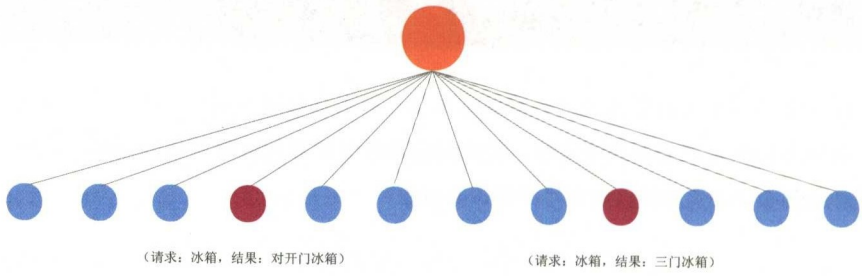


图6-18　广度模型

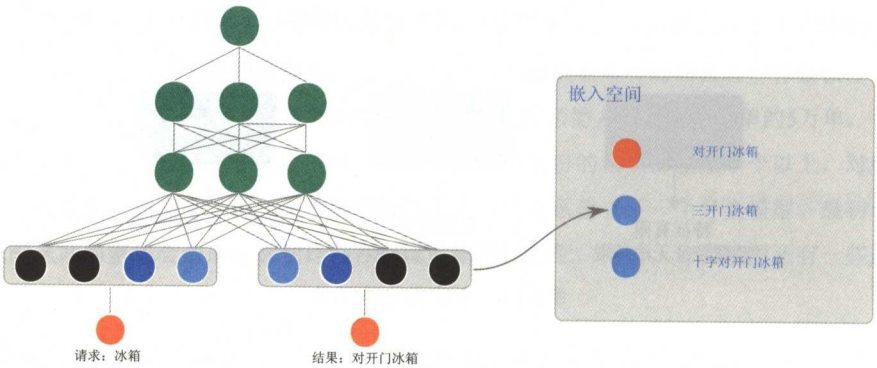


图6-19　深度模型

结合深度与广度的优缺点，我们构建了基于Wide & Deep Learning的排序模型。像“冰箱”这样的请求和“对开门冰箱”这样的物品都属于嵌入特征，用于广度和深度部分。训练的过程中，通过把预测误差回传到两边来训练模型参数。广度模块中的交叉特征变换可以记忆所有稀疏的特定规则，而深度模块则通过嵌入特征实现相似物品的推荐。如图6-20所示。

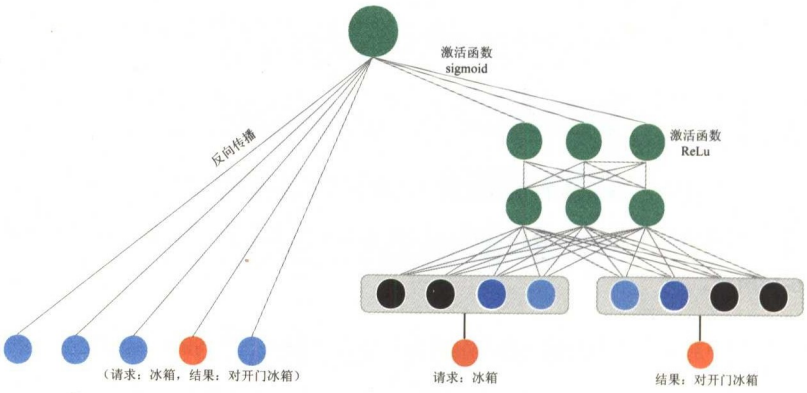


图6-20　列表页排序模型

此外，我们还在Wide & Deep Learning的基础上进行强化学习，以求分类精度的持续提高。例如，用单价作为是否发生购买行为对应的rewards的绝对值，以及使用Q-learning、model-free技术等。

最终，通过线上的AB对比测试，新的列表页排序算法在多个试点类目实现了5%左右的GMV增量和客单价提升，效果大超预期。

6.4 语音识别与客服导航

在人工智能时代，语音交互有可能成为比App触摸交互更好的手机交互方式。以在京东上给手机充话费为例，正常流程是打开手机App，通过导航栏找到手机充值入口，然后输入手机号，再输入充值金额，最后点击确认，整个过程经历了4~5轮交互，总体时间在1分钟以上；而如果换成语音交互方式，则可以是对着京东App说：“给我手机充值100元”，然后系统自动跳转到了支付页面。

各大互联网公司都开始重视语音识别技术，纷纷推出了自己的语音产品，如苹果的Siri、谷歌的Assistant、亚马逊的Alexa，国内的也有科大讯飞、百度语音等。当然，对于京东，我们也急需自己的语音识别技术。语音识别是最具挑战性的机器学习问题之一。国内很长一段时间只有科大讯飞一家公司在研究并成功商业应用。直到2014年，百度经过几年的积累，推出自己的语音识别技术Deep Speech。

语音识别的难度总结起来有三点：算法实现、数据量和训练效率。

在2000年以前，语音识别的核心技术虽然不断涌现，但基本上都停留在研究领域，如混合高斯模型（GMM）、隐马尔科夫模型（HMM）、梅尔倒谱系数（MFCC）机器差分等。在2000年到2010年年间，随着GMM-HMM混合框架的推出，语音识别才开始逐步应用到实际系统中。而语音识别技术的快速发展则是在最近几年，随着深度学习的火热，其在大词汇连续语音识别任务中的良好表现，将语音识别技术推向了一个新的台阶。

京东商城从2016年11月开始研发自己的语音识别引擎，经过1个多月的技术选型，最终决定在Deep Speech 2的基础上，构建更适合电商和客服领域的语音识别技术。Deep Speech 2的优势在于在当前整个语音序列全局优化的基础上，对汉语的声韵母进行了尖峰抽象（建模单元中最具备特征描述能力的一帧语音被抽取出来代表这个声母或者韵母），这样的建模技术

可以轻松混合多种数据源（口音、噪音、远场等）进行训练，不同数据源之间的差异会被抹平。并且，它的解码速度非常快，从2倍实时变成0.05倍实时。

我们的“Sequence-to-sequence encoder-decoder architect”方法在Encoder上使用了Deep CNN，在Decoder上使用了Deep LSTM，然后使用Attention module将Encoder和Decoder连接起来。整体框架如图6-21所示。

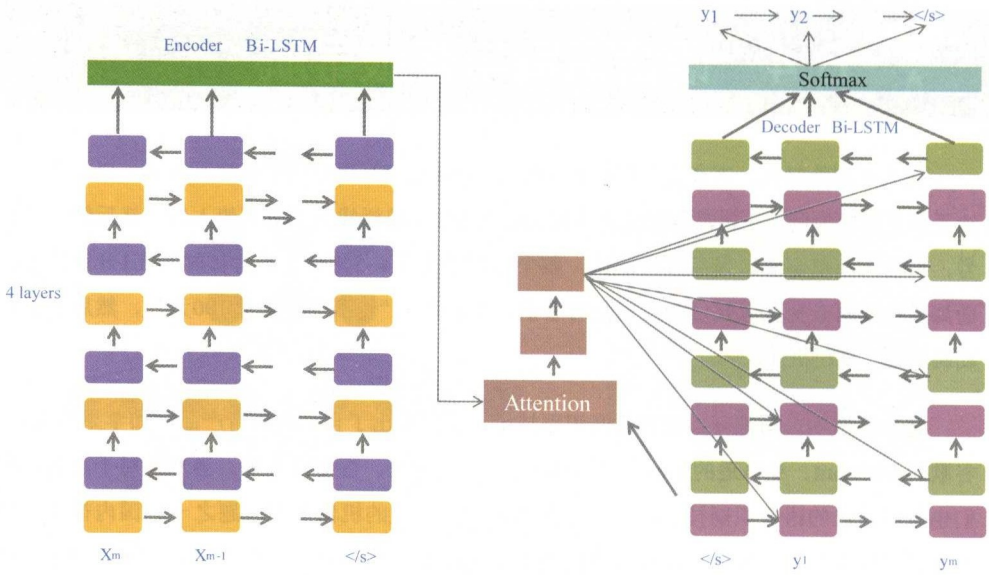


图6-21 端到端语音识别架构

相比较而言，基于CTC的训练语音模型与语言模型的结合比较受限制，只能在Inference阶段进行；而基于Encoder-decoder的架构，能够同时在Train阶段和Inference阶段通过Beam search optimization结合语音模型和语言模型。另外，Encoder-decoder技术之前在机器翻译和图片标注方面有很成功的应用，虽然机器翻译和图片标注没有源序列到目标序列的顺序对应关系，但是语音识别有，因此效果会更好。

数据对语音识别同样相当重要，语音识别要求的数据量通常在1万小时以上，百度更是达到10万小时。纯粹依赖人工标注获得几万小时的数据不大现实，京东的数据来源主要有几种：人工录制、音频切割、数据生成。我们将全国划分为六个区域，每个区域征集了1000人进行录制，即便这样也仅仅获得了3000小时的有效数据。更多的数据则依赖于音频切割和数据生成。互联网上有很多高质量的电台资源或者音视频，可以批量下载做语音训练，但这些数据通常都是一大段长语音，几十分钟，不能直接使用。我们通过声音频率变换识别出静默

区，将大的音频文件切割成一段段4~10秒的小音频，由此也获得了几千小时的有效数据。当然，最大的数据来源还是数据生成。通过对数据进行各种变换，如对原始音频加餐厅、车载、机场等场景噪声，将一段音频里面的声音变快变慢，音量变高变低等，将几千个小时数据集扩展成几万个小时。

虽然GPU性能已经足够强悍，但面对几万小时的语音数据训练也是一个极大的挑战。语音训练对框架的分布式要求非常高，我们最终选用了TensorFlow框架，但对原生的TensorFlow进行了很多深度定制，提高训练速度。深度神经网络训练往往需要对模型和参数进行多次迭代调优，对于数万小时的语音来说，我们最初训练一次需要几周，这样会使得整个项目周期变得相当长，因此我们在软硬件层面上进行了很多优化。在软件层面上，对TensorFlow的Parameter参数更新、网络数据传输还有RNN实现等方面进行了深度定制。在硬件层面上，用Infiniband网络替代传统以太网，包传输延时大大降低，吞吐量提升。最终，训练速度从之前的几周变为几天。

京东的语音识别虽然起步比较晚，但经过半年多的努力，已经取得了快速突破，开始逐步应用在电话客服、商家咚咚、JIMI无人客服等业务领域，用语音替代键盘输入，大大提升了用户体验。以IVR智能语音导航应用为例，语音识别的应用能够免除用户在拨通客服电话之后冗长的播报等待时间，以及在键盘上输入数字的繁琐，直接通过人工智能语音导航迅速对接到处理相应业务的人工客服，实现客户一句话描述问题即可精确匹配对应客服，如图6-22所示。

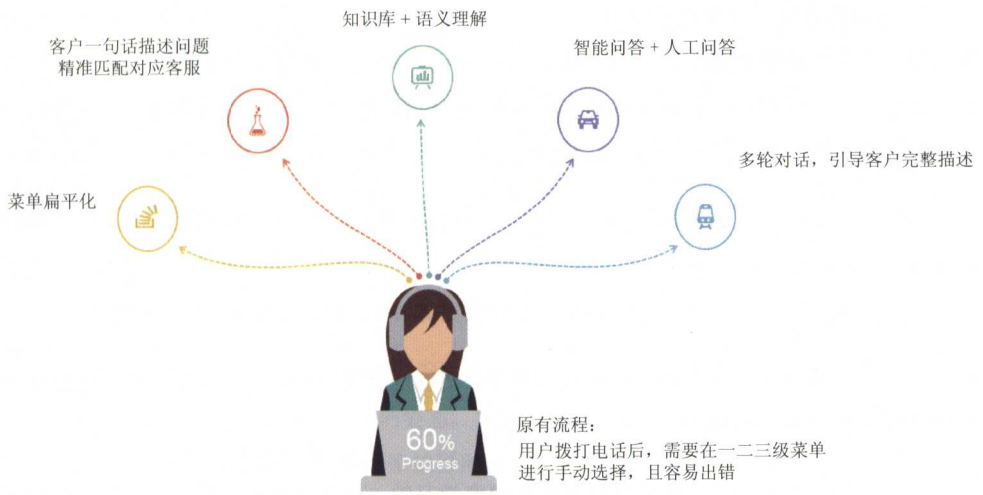


图6-22 智能语音导航应用

6.5 商品上新助手

京东作为国内最具影响力的电商平台之一，保持商品覆盖的全面、与商家建立良好的合作关系、促进有效的商品流通等都是我们的目标。而对于商家的商品与平台间产生联系，上新是第一个环节。目前，京东商城每年上新商品量过亿，且数量在不断增长，就服饰内衣这一品类来讲就具有上新频率高、上新数量多等特点。如图6-23所示，上新流程整体包括商品拍照与修图、商品信息录入与文案准备、商品详情页制作及商品上货等多个环节，流程漫长，对于商家的时间成本和人力成本耗费都很大。

图6-23 上新流程

针对商家的上新流程，利用京东现有技术与数据基础，结合已有的图文不一致校验、电商短文本理解等成果，利用计算机视觉、自然语言处理等深度学习技术，我们为商家打造了智能上新助手，提供全流程服务，促使设计与商业的有效结合。现在，商家仅需要拍照上传，剩下的上新步骤均可以由智能助手完成。

商品信息录入与文案准备阶段，结合现有的商品信息与自然语言处理技术，我们可以针对用户录入的标题和属性进行异常校验，确保商家信息录入的正确性，避免对用户产生误导。此外，我们将根据现有的数据基础，为商家提供有效的商品标题等优化建议，从上新源头为商家的销售赋能，给予商家有效的指导。

同时，根据商品评论数据，我们会提取在评论数据中用户关心的同品类商品问题、同品牌商品问题等，为商家在编辑商品文案环节提供相关的关键词建议，提供最直接的帮助，使商家更好地描述商品信息，同时提升用户浏览商品详情页的效率，提升商品详情页的有用性。

另外，图像识别与自然语言处理技术的结合，以及对图片中信息的有效提取，也能够及时帮助商家快速补全商品信息，提升录入效率。

详情页的制作与输出，是计算机视觉和深度学习的一个非常好的应用。我们使用R-FCN（Region-based Fully Convolutional Networks）+ DCN（Deformable Convolutional Networks）的网络结构进行商品及模特的主体检测。如图6-24所示，R-FCN通过共享RPN（Region Proposal

Network）网络和分类网络的卷积参数能够极大减少对ROI（Region-of-Interest）进行分类的计算量，在保证准确率的同时，能够明显提升训练和测试阶段的速度。

另外，经典的CNN（Convolutional Neural Networks）网络具有固定的几何结构，对几何形变物体的建模具有局限性。DCN通过引入deformable convolution和deformable RoI pooling两个新模块很好地解决了这个问题。如图6-25所示的结构，这两个模块通过加入可学习的二维偏移量（offset）改变了卷积过程中的固定采样点，从而对可形变的复杂物体检测具有很好的效果。

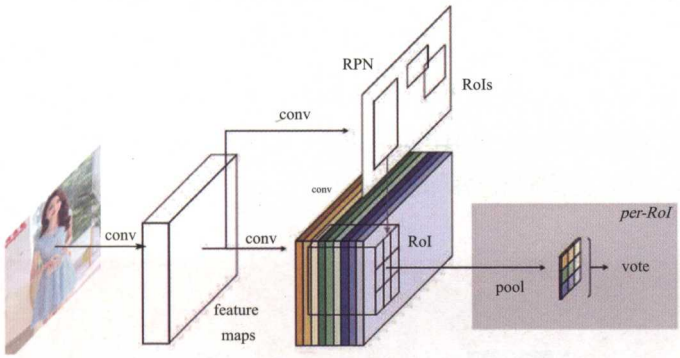


图6-24 R-FCN网络结构示意图

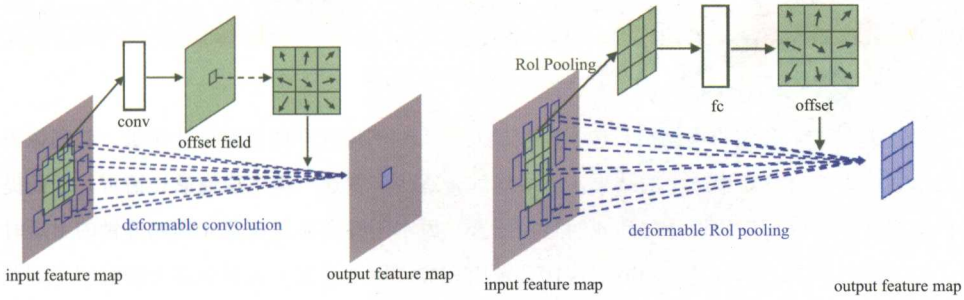


图6-25 R-FCN的两个新模块

通过以上网络结构，我们能够准确识别并理解图片，包括识别图片里的商品分类、商品主体、颜色、细节及相关属性，模特的姿态、面向、动作等内容，如图6-26所示。同时，基于图片大数据的分析，我们能够给到商家拍摄的建议，和图片选择的建议。往往商家上架一款商品会拍摄大量图片，如何进行图片的优中选优，如何选择最吸引用户的商品主图，如何进行有效的图片排列等问题，智能上新助手都能给商家最有效的答案。

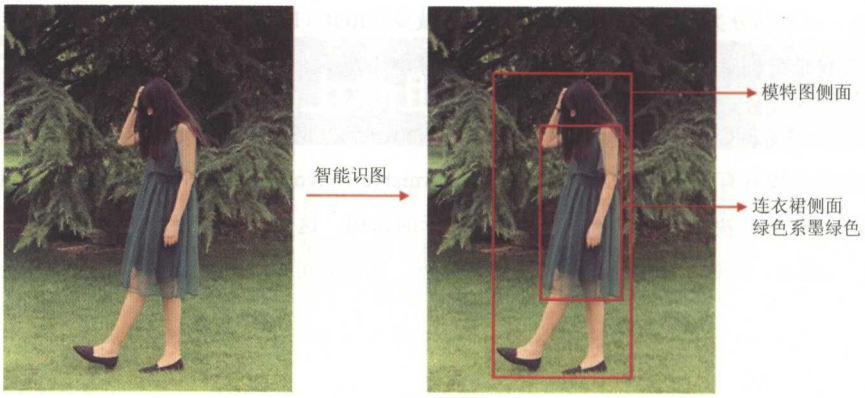


图6-26　智能识图

在图像识别的基础上，我们能够完成批量修图、智能裁剪等图像处理功能，从而保证将最精致的商品图片呈现给用户，并且能够完成批量操作，避免商家的大量重复性工作，提升整体的上新效率。如图6-27所示，为一键上新的流程。

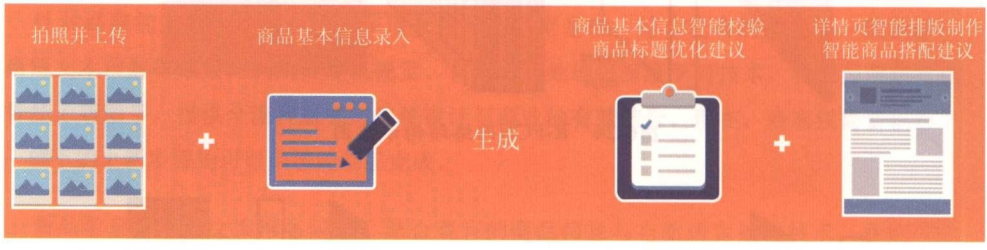


图6-27　一键上新的流程

在此基础上，我们将根据用户选择的模板及模板规则进行详情页的设计排版，将图像识别的结果、自然语言处理产生的结果，有效地综合运用，快速一键生成商品详情页，从而提升视觉表现，实现真正的设计与商业的有效结合。原本一件商品要花费60到90分钟的上新时间，而智能上新助手则可以帮助商家在10分钟内完成；原本商家一天甚至多天的时间全部花费在了商品上新环节上，现在可以有更多的时间去打磨与销售自己的商品；原本枯燥无味又繁琐的上新流程，将会变得有趣而有效。

为了回馈社会，我们将书中提到的一些开源项目放在GitHub上供大家下载研究，也希望能和大家一起讨论探索，将项目持续推进，做得更好，欢迎大家关注！

开源项目地址：https://github.com/ipdcode。

或扫描二维码：

Broadview®
www.broadview.com.cn
博文视点 · IT出版旗舰品牌
技术凝聚实力 · 专业创新出版

京东
基础架构建设之路

基础架构是京东业务的技术基石。本书的作者们，作为过去几年里推进京东基础架构变革的技术实践者，一直坚定执行京东集团“下一个十二年，只有技术”的发展路线。借《京东基础架构建设之路》这本书，我代表所有奋战在一线的技术研发团队，将京东在基础架构技术领域这几年中的发展和创新分享给关注我们的朋友。感谢所有互联网技术从业者对我们的关注、帮助与指正。

马松
京东集团高级副总裁、商城研发体系负责人

《京东基础架构建设之路》这本书，从底层的容器管理集群技术，到服务框架、分布式内存数据库和分布式文件存储系统，再到机器学习在京东的多场景应用和商品数据知识图谱的构建，都做了详细的介绍，向大家展现了整个系统搭建的发展历程。同时，书中也解密了京东技术研发在每年618和双11超大流量和高并发时刻的应对策略，相信会对互联网和电商行业的从业者有着不错的借鉴作用。

陈恩红
中国科学技术大学计算机学院教授、博士生导师、副院长

博文视点Broadview
京东图书

策划编辑：符隆美
责任编辑：张春雨
封面设计：李　玲
特约监制：杨海峰　特约策划：吴迪

上架建议：计算机>基础架构

ISBN 978-7-121-32865-7
9 787121 328657 >
定价：79.00元